Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Vitek

Agenti za simulacijo človeškega obnašanja v svetu igre

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: izr. prof. dr. Peter Peer

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Agenti za simulacijo človeškega obnašanja v svetu igre

Simulating human-like behaviour in games using intelligent agents

Tematika naloge:

Implementirajte in med seboj primerjajte različne inteligentne agente, ki bodo v svetu igre reševali nalogo. Najprej naredite teoretični pregled agentov. Nato sestavite ustrezen scenarij za testiranje inteligentnosti agentov ter izberite primerno mero za analizo rezultatov. Programski izdelek naj simulacijo poganja v 3D svetu ter ima možnost vizualizacije predstavitve sveta za gibanje agenta.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matej Vitek sem avtor diplomskega dela z naslovom:

Agenti za simulacijo človeškega obnašanja v svetu igre

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Petra Peera,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. septembra 2015 Podpis avtorja:

Na tem mestu bi se rad najprej zahvalil mentorju izr. prof. dr. Petru Peeru za potrpežljivost in za čas, ki si ga je vzel za pomoč pri tem diplomskem delu. Poleg tega bi se zahvalil izr. prof. dr. Poloni Oblak za pomoč pri izbiri teme in iskanju mentorja ter as. mag. Cirilu Bohaku za pomoč pri težavah z razvojnim okoljem Unity. Nazadnje pa bi se zahvalil še očetu za premnoge nasvete in mami za večno spodbudo pri izdelavi diplomske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
TDA	table-driven agent	agent s tabelo
SRA	simple reflex agent	preprost odzivni agent
SBA	state-based agent	agent s stanji
MBA	memory-based agent	agent s spominom
GBA	goal-based agent	ciljno usmerjen agent
UBA	utility-based agent	agent s funkcijo koristnosti
LA	learning agent	učljiv agent
AEA	autonomous exploration agent	agent z avtonomnim raziskovanjem
NPC	non-player character	računalniško voden lik
RTS	real-time strategy	realno-časovna strateška igra
RPG	role-playing game	igra igranja vlog
OTS	over-the-shoulder	čez ramo
BDI	belief-desire-intention	prepičanje-želja-namen
NN	nearest neighbour	najbližji sosed
TSP	travelling salesman problem	problem potujočega trgovca

Povzetek

Področje umetne inteligence v zadnjem času zelo napreduje, pristopi področja pa se uporabljajo tudi za računalniško vodenje likov v svetu igre. Pojem inteligentnih agentov nam omogoči dobro teoretično podlago za izbiro najprimernejših pristopov za dosego racionalnega obnašanja teh likov. Z dodatkom omejitev na agentove sposobnosti pa dobimo res človeku podobno vedenje. V diplomskem delu smo predstavili in med seboj primerjali različne tipe agentov, ki se v igrah (a ne le v igrah) v ta namen uporabljajo ter predstavili, kako se omejitev sposobnosti implementira v svet igre. Cilj diplomskega dela je prikazati dobre in slabe lastnosti vsakega od agentov in ugotoviti, za izvedbo kakšnih nalog je primeren. Pokazali smo, da so že preprosti agenti lahko povsem primerni za določene naloge, za bolj zahtevne pa agenti pogosto potrebujejo bolj napredne pristope. Ugotovili smo, da na obnašanje najbolj vpliva dodatek ciljne usmerjenosti agentov, poleg nje pa je v veliko pomoč tudi pristop avtomata s stanji.

Ključne besede: agent, inteligenca, umetna inteligenca, igre, svet igre, razvoj iger.

Abstract

The field of artificial intelligence has come a long way in the past 50 years, and studies of its methods soon expanded to a field in which they are of great practical value – computer games. The concept of intelligent agents provides a much needed theoretical background for the comparison of various different approaches to intelligent, rational behaviour of computer-controlled characters in games. By combining rationality with certain limitations of the capabilities of our agents, we can achieve very human-like behaviour. In this diploma thesis we introduced and compared various types of agents that are used in games (but not only in games) and showed how to implement meaningful, reasonable limitations to agent capabilities into the game world. The aim of this thesis is to show the strengths and weaknesses of each type of agent and decide what types of tasks it is suitable for. We showed that even the simplest agents can succeed in their tasks in certain task environments, while more difficult task environments often require a more advanced agent architecture. The addition of goals into the agent architecture had the biggest impact on the agent's behaviour, while the finite-state machine approach kept our implementation simple and compact.

Keywords: agent, intelligence, rationality, artificial intelligence, games, game world, game development.

Poglavje 1

Uvod

Področje umetne inteligence je danes izjemno obsežno. Umetna inteligenca se uporablja v gospodarske namene, za reševanje matematičnih problemov, za zdravstvene diagnoze, za igranje šaha ipd. V tej nalogi bomo obravnavali pristope umetne inteligence, ki se uporabljajo v svetu iger.

V ta namen bomo implementirali in med seboj primerjali različne inteligentne agente, ki bodo v danem svetu morali reševati določene naloge. Uporabili bomo programsko orodje Unity3D, namenjeno razvoju iger in simulacij v 2D in 3D svetovih [9].

1.1 Kaj sploh je inteligentni agent?

Poglejmo pomen vsake besede posebej.

V računalništvu pojem **agent** označuje računalniški program, ki v danem okolju (virtualnem ali resničnem) deluje avtonomno. Od okolja s senzorji sprejema informacije v obliki zaznav in se na njih odziva z akcijami, s katerimi vpliva na stanje okolja.

Pojem **inteligence** oz. **racionalnosti** pa tukaj pomeni, da agent vedno izbira akcije, ki ga peljejo v čim boljše stanje (v primeru, da imamo negotovost, pa v čim boljše pričakovano stanje), glede na njegovo trenutno znanje.

Za bolj formalno definicijo uvedemo **mero uspeha** (angl. *performance*

measure), ki predstavlja oceno tega, kako dobro se je agent pri opravljanju nalog izkazal. Način ocenjevanja je seveda odvisen od okolja in nalog, zato navadno to mero definiramo sami, ko te naloge določamo. [28] S pomočjo te mere potem dobimo formalno definicijo inteligentnega (racionalnega) agenta:

> "Za vsako možno zaporedje zaznav inteligentni agent izbere akcijo, ki maksimizira pričakovano mero uspeha, glede na podatke iz zaznav in agentovo predznanje." [32]

Pomembno pa je tukaj omeniti, da to ne pomeni, da mora agent dejansko poznati in v implementaciji uporabljati svojo mero uspeha. Kot bomo videli v 3. poglavju, se lahko tudi zelo preprosti agenti, ki o svoji meri uspeha ne vedo ničesar, pri določenih nalogah obnašajo racionalno.

1.2 Svet igre

Svet igre je okolje z določenimi pravili in zakoni, ki veljajo za osebke, ki se v njem nahajajo. Ta pravila niso nujno za vse enaka, čeprav vsaj fizikalni zakoni ponavadi so. Osebke lahko upravljajo ljudje, lahko pa jih upravlja računalnik (angl. *non-player character – NPC*). Osebki lahko med seboj komunicirajo, si pomagajo, se napadajo, si izmenjujejo poteze itd. To so z vidika agenta primeri akcij, s katerimi lahko vpliva na okolje in na druge agente.

Opozorimo, da pojma NPC in inteligentni agent ne predstavljata iste stvari. NPC je lik, ki je v igri viden (ima neko telo), implementacija inteligentnega agenta pa je le način, da tak lik upravljamo. Poleg tega pojem inteligentnega agenta sam po sebi zahteva inteligentno odločanje in pogosto tudi avtonomijo, NPC pa zahteva le videz inteligence in avtonomije. [21]

V svetu iger od agentov poleg inteligence, kot smo jo definirali v razdelku 1.1, zahtevamo še eno lastnost: poštenost. Računalnik ima seveda v vsakem trenutku dostop do vseh informacij o svetu, vendar pa idealno želimo, da za upravljanje posameznega agenta uporablja le tiste, ki jih lahko

ta agent v tistem trenutku smiselno zazna. Zato agentom omejimo znane informacije in tako pravzaprav programsko nadomestimo oziroma omejimo senzorje. Tako agenti ne vidijo skozi stene in ne slišijo šuma na ogromni razdalji, poleg tega pa ima vsak od njih svoje zaznave in nima direktnega dostopa do zaznav preostalih agentov. [30]

1.3 Struktura naloge

V 2. poglavju natančno definiramo, kaj je scenarij, kako ga opišemo v kompaktni in natančni obliki in nato opišemo nekaj lastnosti, ki za scenarije lahko veljajo in nam pomagajo pri izbiri primernih agentov za njih.

V 3. poglavju naredimo formalno delitev inteligentnih agentov glede na kompleksnost in sposobnosti ter vsakega posebej opišemo. Povemo za kakšne scenarije je kateri primeren in kakšne lastnosti imajo.

V 4. poglavju definiramo testna scenarija, ki ju bomo uporabljali za primerjavo agentov in opišemo njune lastnosti. Pri tem sledimo definicijam iz 2. poglavja.

V 5. poglavju se lotimo najprej implementacije vida in nato agentov. Najprej pojasnimo izbiro agentov, ki jo bomo za posamezen scenarij uporabili, nadaljujemo pa z implementacijo vsakega agenta posebej. Pri prikazu delovanja si pomagamo s psevdokodo in diagrami stanj.

V 6. poglavju primerjamo agente glede na to, kako dobro so delovali. V tabelni obliki izpišemo povprečne rezultate meritev in jih nato komentiramo in analiziramo.

V 7. poglavju naredimo pregled celotne naloge in premislimo, kaj bi se še dalo izboljšati oz. nadgraditi (nadaljnje delo).

1.4 Licenca

Izvorna koda programa je objavljena na spletu, na repozitoriju GitHub [3]. Zaščitena je pod licenco GNU GPL v3 [4]. Naš cilj je, da omogočimo nadaljnji

razvoj in uporabo naše programske opreme. Odprtokodna licenca omogoča posameznikom, da program prilagodijo svojim potrebam. Na spletnem repozitoriju je možno tudi podajanje predlogov za izboljšave ter dodatne funkcionalnosti. Tako lahko tudi raziskovalci brez programerskega znanja (ki so ciljni uporabniki programa) prispevajo k projektu. Program je testiran v operacijskem sistemu Windows, prenos kode na druge operacijske sisteme pa ne bi smel biti problematičen, če le-ti podpirajo programsko orodje Unity [12].

Poglavje 2

Scenarij

Za opis scenarija (angl. *task environment*) uporabimo opis **PEAS** [32] (angl. *performance, environment, actuators, sensors*). Za popoln opis moramo torej vedeti, kaj točno od agenta želimo, v kakšnem svetu bo deloval, na kakšen način lahko vpliva na okolje in nazadnje, kako od okolja pridobiva informacije.

Za primer si poglejmo šah. Šah je dober primer preprostega scenarija, medtem ko so agenti, ki jih uporabljamo za igranje šaha, dandanes vse prej kot preprosti.

Uspeh (angl. *performance*) lahko definiramo opisno (v šahu želimo, da agent doseže najboljši možni izid – če se le da, zmago), lahko pa že direktno definiramo kar mero uspeha.

Mera uspeha je pravzaprav funkcija, ki jo bo agent poskušal maksimizirati. V šahu jo torej lahko definiramo zelo preprosto v stilu turnirskih iger, torej ena točka za zmago, pol točke za remi in nič točk za poraz. Tako bo inteligentni agent vedno izbiral poteze, ki ga vodijo k najboljšemu še možnemu izidu (če npr. zmaga ni več mogoča, bo poskušal doseči remi). Taka definicija mere uspeha je dobra, če opazujemo vsako odigrano partijo posebej.

Paziti moramo, da pri testiranju in ocenjevanju vedno uporabimo dovolj obsežen (in raznolik) vzorec nasprotnikov in iger, saj so sicer lahko specifični agenti zavajajoče dobri (npr. agent, ki zna igrati samo belo stran ali pa agent, ki se zna braniti samo s sicilijansko obrambo).

Omenimo še, da pri načrtovanju agenta za igranje šaha uspeh lahko definiramo tudi drugače – lahko želimo, da agent sicer predstavlja izziv, ni pa nepremagljiv (za nasprotnike z neko določeno stopnjo šahovskega znanja). Temu primerno bi potem lahko definirali mero uspeha glede na želeno porazdelitev zmag, remijev in porazov in za testiranje uporabili primeren vzorec nasprotnikov s primernim šahovskim znanjem. Lahko želimo tudi, da agent vsebuje nekaj naključnosti, da torej npr. ne igra vedno iste otvoritve.

Na tak način lahko tudi dodatne lastnosti, ki jih pogosto zahteva svet igre (nepopolnost, premagljivost, nepredvidljivost ...), uporabimo kot del mere uspeha, kar nam omogoča, da uporabimo obstoječo definicijo inteligentnega agenta tudi za reševanje teh problemov.

Lep primer evaluacije agentov je Turingov test, ki se v osnovni obliki uporablja za agente, ki komunicirajo s človekom. V testu nas zanima, ali lahko človek iz komunikacije ugotovi, ali je na drugi strani računalniško voden agent ali človek. V osnovi gre sicer za precej subjektiven test, vendar pa se da s testiranjem na dovolj udeležencih definirati tudi mero uspeha kot npr. delež udeležencev, ki so verjeli, da se pogovarjajo z računalnikom. Z nekoliko bolj kompleksnim pristopom so se tega problema lotili Brandao, Reis in Rocha. [16]

Okolje (angl. *environment*) je svet, v katerem agent deluje. Lahko je resnično ali virtualno – lahko gre za nek kraj v resničnem svetu, lahko gre za množico spletnih strani, lahko gre za virtualen ali pa fizičen labirint, skozi katerega se mora agent prebiti. Okolje ima različna stanja, na katera lahko agenti vplivajo. Stanje okolja vključuje tudi stanja posameznih agentov, ki se v njem nahajajo.

Pri razvoju agenta temu konceptu rečemo tudi **svet** (angl. *agent world, world of operation*).

V šahu okolje predstavlja šahovnica s figurami, pozicija figur na šahovnici pa predstavlja stanje okolja.

Prožila (angl. *actuators*) so sredstva, s katerimi agent lahko vpliva na stanje okolja. V virtualnih okoljih nas tu zanima le, katere akcije so agentu

na voljo. Možne akcije so seveda lahko odvisne od trenutnega stanja okolja.

V šahu ima agent možnost premakniti figuro (v skladu s šahovskimi pravili), čakati nasprotnika (v virtualnem primeru te akcije morda ne potrebujemo – odvisno od implementacije celotne aplikacije) ali pa se predati (to akcijo mu lahko omogočimo ali pa tudi ne). Če po šahovskih pravilih nima možnih potez, potem nima na voljo tudi nobene akcije, igra pa se nemudoma zaključi z remijem.

Če imamo agenta, ki šah igra na fizični šahovnici, torej v resničnem svetu, pa moramo tu opisati tudi, kako bo dejansko figure premikal (npr. z robotsko roko) in kako bo oznanil predajo. V primeru, ko nima na voljo možnih potez, mu lahko dodamo tudi akcijo, s katero nasprotnika na to opozori (v virtualnem primeru bo za to poskrbela že aplikacija).

Senzorji (angl. *sensors*) so sredstva, s katerimi agent prejema informacije iz okolja. Pri šahu je v virtualnem primeru to lahko direktna informacija o poziciji figur (npr. v obliki tabele velikosti 8×8), v primeru fizičnega šaha pa lahko uporabimo kamero in programsko opremo za razpoznavo pozicije iz slike.

2.1 Lastnosti scenarija

Scenarij lahko opišemo z nekaj dodatnimi lastnostmi, ki nam bodo kasneje pomagale pri izbiri primernega agenta za izpolnjevanje nalog. Scenarije bomo ločili glede na to, ali so:

- popolnoma vidni, delno vidni, ali nevidni,
- večagentni ali enoagentni,
- deterministični ali stohastični,
- epizodni ali sekvenčni,
- statični ali dinamični,

- diskretni ali zvezni,
- poznani ali nepoznani.

Scenarij je **popolnoma viden** (angl. *fully observable*), če ima agent preko senzorjev vedno dostop do informacije o celotnem stanju okolja. Če lahko zazna le del stanja, je scenarij **delno viden** (angl. *partially observable*). To, kateri del stanja lahko zazna, je lahko odvisno od stanja samega. Če agent nima (uporabnih) senzorjev, torej nikoli ne more zaznati ničesar, je scenarij **neviden** (angl. *unobservable*).[1] Scenarij navadno obravnavamo kot popolnoma viden tudi v primeru, ko agent lahko v vsakem trenutku zazna vse informacije o stanju okolja, ki so pomembne za izbiro naslednje akcije.

V delno vidnih scenarijih bomo pogosto potrebovali agenta, ki si bo znal zapomniti ali pa uganiti informacije o delih okolja, ki jih v nekem trenutku ne bo zaznaval. V računalniških igrah so scenariji, kot smo omenili v razdelku 1.2, popolnoma vidni, vendar pa jih preko programske simulacije senzorjev večinoma spremenimo v delno vidne.

V šahu je scenarij popolnoma viden, saj v vsakem trenutku agent lahko dobi celovito informacijo o poziciji figur na šahovnici. Če pa v postopek izbire poteze vključimo tudi informacijo o tem, kaj naš nasprotnik trenutno razmišlja, pa je ta scenarij delno viden, saj nasprotnikovih misli agent s senzorji ne more zaznati. V tem primeru bomo seveda v implementaciji agenta potrebovali neko sredstvo za ugibanje nasprotnikovih misli (npr. strojno učenje).

Včasih moramo tudi teoretično popolnoma vidne scenarije v praksi obravnavati kot delno vidne zaradi različnih omejitev. Kot primer vzemimo agenta, ki deluje na množici spletnih strani. V teoriji sicer v vsakem trenutku lahko dostopa do katerekoli od njih, vendar pa zaradi omejitev spletne povezave ter računskih zmožnosti procesorja in pomnilnika v resnici nikoli nima ažurne informacije o celotnem stanju. Zato moramo tak scenarij obravnavati kot delno

[1] Pogosto sta v literaturi rabljena tudi izraza **dostopen** (angl. *accessible*) in **nedostopen** (angl. *inaccessible*). Prvi je ekvivalenten popolni vidnosti, drugi pa združuje delno vidne in nevidne scenarije.

viden.

Scenarij je **večagenten** (angl. *multiagent*), če v njem deluje več vzajemno delujočih agentov, torej agentov, katerih mere uspeha so medsebojno odvisne (posledično je medsebojno odvisno tudi obnašanje agentov). Ta odvisnost je lahko vedno pozitivna (torej povečanje mere uspeha enega pomeni tudi povečanje mere uspeha drugega) – v tem primeru je scenarij sodelovalen (angl. *cooperative*). Lahko je vedno negativna (povečanje mere uspeha prvega pomeni padec mere uspeha drugega) – tedaj imamo tekmovalen (angl. *competitive*) scenarij. Lahko pa je včasih pozitivna, včasih pa negativna – mešan scenarij. Tukaj pogosto kot agente štejemo tudi igralce. Tako je na primer scenarij, ko imamo igralca in agenta, ki se morata skupaj prebiti skozi množico ovir, z vidika tega agenta dvoagenten.

Scenarij je **enoagenten** (angl. *single agent*), če v njem deluje le en agent. Scenarij z več agenti pa je enoagenten z vidika posameznega agenta, če lahko preostale agente obravnava kot le del sveta, ne pa kot osebke, ki poskušajo maksimizirati svojo mero uspeha, ki je odvisna tudi od agenta. Preprosteje rečeno, scenarij obravnavamo kot enoagenten, če mera uspeha posameznega agenta ni odvisna od obnašanja preostalih. Pri tem ne upoštevamo preprostih interakcij z zelo majhnim vplivom, kot je npr. izmikanje drugim agentom pri iskanju poti, ki bo rahlo vplivalo na agentovo hitrost.

V večagentnih scenarijih pogosto kot inteligentni izbiri akcij nastopata komunikacija (predvsem v sodelovalnih scenarijih) in naključno vedenje (predvsem v tekmovalnih scenarijih).

Šahovski scenarij je večagenten tekmovalen – mera uspeha enega igralca je obratno sorazmerna z mero uspeha drugega.

Scenarij je **deterministično** (angl. *deterministic*), če je naslednje stanje okolja natanko določeno s trenutnim stanjem in izbiro akcije. Pri tem v večagentnih scenarijih ne upoštevamo sprememb, ki jih povzročijo akcije ostalih agentov. Če pa imamo pri kateri od akcij, ki so na voljo, možnih več izidov (ki imajo ponavadi določene verjetnosti), pa je scenarij **stohastičen** (angl. *stochastic*).

V deterministič nem scenariju agentu ni treba razmišljati o negotovosti, razen v primeru delno vidnega scenarija, kjer lahko pride do negotovosti o stanju tistega dela okolja, ki ga trenutno ne zaznava (če si ga ni kdaj prej zapomnil).

V šahu je scenarij deterministič en, saj lahko vedno določimo novo pozicijo figur, če poznamo prejšnjo in vemo, katero potezo smo izvedli. V primeru v resničnem svetu pa bi lahko rekli, da je scenarij stohastič en, saj se lahko nekdo po nesreči zaleti ob mizo in prevrne figure na šahovnici.

Scenarij je **epizoden** (angl. *episodic*), če izbira akcije ni odvisna od prejšnjih akcij in ne vpliva na prihodnje akcije. V nasprotnem primeru je scenarij **sekvenč en** (angl. *sequential*).

V epizodnem scenariju agent ne potrebuje zmožnosti planiranja. Vendar pa je v igrah večina scenarijev sekvenč nih. To velja tudi za šah, kjer poteze seveda vplivajo na to, kaj se bo dogajalo kasneje v partiji (če na primer ne izvedemo rošade, bomo kasneje morali pogosto igrati poteze za branjenje kralja).

Scenarij je **statič en** (angl. *static*), če se stanje okolja ne spreminja medtem ko agent razmišlja o naslednji akciji. Sicer je **dinamič en** (angl. *dynamic*).

V statič nih scenarijih nam ni treba skrbeti za to, kako hitro se agent odloči za naslednjo akcijo (razen seveda z vidika uporabniške izkušnje), poleg tega pa mu med odločanjem ni treba paziti na spremembe stanja okolja.

V šahu je scenarij statič en, če ne upoštevamo igralne ure – le-ta bo tekla medtem ko se agent odloča o naslednji potezi, pozicija figur pa bo ostala enaka.

Stanja okolja, čas ter agentove zaznave in akcije imajo lahko **diskretno** (angl. *discrete*) ali pa **zvezno** (angl. *continuous*) zalogo vrednosti.

V zveznih scenarijih pogosto uporabljamo diskretne približke, ki so bolj primerni za računalniške postopke odločanja. Kljub temu pa senzorje, ki te zvezne podatke zaznavajo in predstavljajo v računalniku prijazni obliki, štejemo med zvezne.

V šahu (spet brez igralne ure) so nabori vseh štirih množic diskretni – stanj okolja je končno mnogo, čas merimo v potezah (ki so sicer tudi omejene, vsekakor pa so diskretne – označimo jih lahko z naravnimi števili), različnih zaznav je prav toliko kot stanj okolja (to vedno drži v popolnoma vidnih scenarijih, ne bi pa držalo na primer, če bi stanje vključevalo tudi zgodovino potez), število akcij pa je prav tako neko (navzgor omejeno) naravno število.

Pravimo, da je scenarij **poznan** (angl. *known*), če vnaprej poznamo pravila in zakone, ki v njem veljajo. Bolj natančno, v poznanem scenariju zahtevamo, da agent vedno lahko predvidi vpliv, ki ga bo njegova akcija imela na stanje okolja. V nasprotnem primeru je scenarij **nepoznan** (angl. *unknown*).

Jasno je, da so nepoznani scenariji z vidika razvoja agentov zelo zapleteni, saj takoj zahtevajo zmožnost učenja in prilagajanja. K sreči pa v igrah večinoma poznamo pravila, torej je večina scenarijev, s katerimi se bomo ukvarjali, poznanih.

V šahu se seveda spoznamo s pravili preden začnemo razvijati agenta, zato je ta scenarij poznan.

Poglavje 3

Pregled agentov

Obstaja nekaj različnih pristopov k arhitekturi implementacije agentov. V tej nalogi bomo sledili delitvi, ki sta jo konkretizirala Russel in Norvig [32] in se v večini literature uporablja kot osnovna delitev inteligentnih agentov, z dodatkom enega tipa agenta – agenta s stanji. Pri tej delitvi gre v osnovi za postopno nadgrajevanje agentov, čeprav se nekateri lahko uporabljajo tudi brez lastnosti prejšnjih.

Naloga agenta je, da se iz svojih zaznav na vsakem koraku odloči za najboljšo akcijo. Postopek izbire akcije najlažje definiramo kot matematično funkcijo – seveda samo konceptualno, za dejansko implementacijo tak pristop ni primeren, kot bomo videli v razdelku 3.1.

Z $\mathcal{Z}$ označimo množico vseh možnih zaporedij agentovih zaznav. Ta bo končna, če imamo končno množico zaznav in omejen čas delovanja agenta. Z $\mathcal{A}$ označimo množico možnih akcij. Potem lahko agentovo **odločitveno funkcijo** definiramo kot

$$f : \mathcal{Z} \to \mathcal{A} \tag{3.1}$$

Ta funkcija vsakemu možnemu zaporedju zaznav priredi ustrezno akcijo. Na ta način dobimo dober koncept za delovanje agenta – na podlagi dosedanjih zaznav se odloči, katero akcijo bo izbral.

Pri posameznih agentih bomo podali še nekaj primerov iz računalniških iger, kjer so le-ti uporabljeni ali pa bi jih lahko uporabili. Pri tem mo-

ramo opozoriti, da se lahko konkretna implementacija agentov v teh igrah močno razlikuje od take, kot bi jo pričakovali pri tem tipu agenta, saj lahko uporablja kakšno posebno arhitekturo, drug tip programskega jezika (npr. logične programske jezike), se prilagaja omejenim računskim zmogljivostim ipd. Osredotočamo se torej bolj na obnašanje agenta kot na samo implementacijo.

3.1 Agent s tabelo

Agent s tabelo (angl. *table-driven agent – TDA*) je direktna implementacija odločitvene funkcije po enačbi (3.1). V tabeli ima za vsako možno zaporedje zaznav shranjeno primerno akcijo, ki jo na vsakem koraku preprosto prebere iz te tabele, glede na trenutno zaporedje zaznav.

V praksi seveda tak agent ni uporaben, saj je število vnosov v tabeli enako $|\mathcal{Z}|$. To število je ogromno (pogosto celo neskončno) tudi v najpreprostejših okoljih, sploh če nimamo časovne omejitve, tabelo pa moramo stalno hraniti v pomnilniku. Poleg tega moramo, če akcije niso preprosto izračunljive iz zaporedja, to tabelo napolniti sami, kar pa je seveda veliko preveč dela za programerja. Tako bo na primer v igri križci in krožci (pretirano preprosto okolje) tabela imela približno 3×10^5 vnosov (oz. dvakrat toliko, če lahko začnemo s križci ali s krožci), v šahu več kot 10^{150}, v svetu ene najstarejših računalniških iger Pong pa bo že neskončna (igra namreč ni časovno omejena, torej je zaporedje zaznav lahko poljubno dolgo).

Vseeno pa je agent s tabelo dober prvi korak, saj konceptualno prikaže idealno obnašanje agenta – agent bo za vsako možno zaporedje zaznav v konstantnem času (predpostavimo, da je dostop do tabele možen v konstantnem času) izbral najboljšo možno akcijo.

3.2 Preprost odzivni agent

Preprost odzivni agent (angl. *simple reflex agent – SRA*) je najpreprostejši agent, ki se v praksi dejansko uporablja. Ravna se po preprostih če-potem pravilih, ki jih vnaprej predpiše programer.

Preprost odzivni agent za izbiro akcije vedno uporablja le trenutno zaznavo, zato je bolj primeren za popolnoma vidne scenarije. Sicer tudi v določenih delno vidnih lahko zanesljivo opravi nalogo, vendar pa navadno manj učinkovito kot kakšen boljši agent. Večinoma pa bo v delno vidnih scenarijih odpovedal (pogoste so neskončne zanke). To težavo lahko nekoliko olajšamo, če v izbiro akcije vpeljemo nekaj naključnosti.

Tak agent seveda tudi nima možnosti planiranja in si ne zapomni svojih akcij, torej se bo slabše odrezal v sekvenčnih scenarijih – to pa ne pomeni, da naloge v le-teh ne bo opravil, le da jo bo opravil slabše kot agent, ki to možnost ima.

Preprost odzivni agent ima torej zelo omejeno zmogljivost, je pa zelo preprost za implementacijo in deluje zelo hitro.

V razdelku 3.1 smo omenili, da je realizacija agenta s tabelo že v svetu igre Pong (1972) [6] tudi teoretično nemogoča, saj potrebuje neskončno tabelo. Realizacija preprostega odzivnega agenta zanjo pa je bolj ali manj trivialna – njegovo odločitveno funkcijo lahko vidimo na sliki 3.1. Implicitno v primeru, ko je lopar že v primerni poziciji, ne izvedemo nobene akcije – tudi to je lahko povsem inteligentna izbira.

Spoznali smo torej, da je že preprost odzivni agent v praksi mnogo boljši pristop kot agent s tabelo.

Čeprav tega morda ne bi pričakovali, pa se preprosti odzivni agenti pogosto uporabljajo tudi v novejših igrah. Dober primer je igra Dragon Age: Origins (2007) [2], kjer so številni mimoidoči pravzaprav preprosti odzivni agenti – naključno se premikajo po nekem vnaprej določenem območju (ali pa celo stojijo pri miru) in igralcu, če jih le-ta ogovori, odgovorijo z naključno izbranim izmed nekaj vnaprej predpisanih odgovorov. Podobne like lahko srečamo tudi v igrah serije Witcher. Preprosti odzivni agenti so torej pogosti

Slika 3.1: Odločitvena funkcija SRAja za igro Pong.

Input: current_percept
Output: action

---

if BallAbovePaddle(current_percept) **then**
    return *move up*
else if BallBellowPaddle(current_percept) **then**
    return *move down*

v igrah igranja vlog (angl. *role-playing game – RPG*) in podobnih virtualnih svetovih in pomagajo pri občutku realizma in ambientu. Kot bomo videli v sledečih razdelkih, v igrah RPG srečamo skoraj vse vrste agentov.

Preprosti odzivni agenti so pogosti tudi v (predvsem starejših) dirkaških igrah (Need for Speed, F1 ipd.), saj je vožnja dirkalnika v svetu teh iger dovolj preprosta.

3.3 Agent s stanji

Agent s stanji (angl. *state-based agent – SBA*) je končni avtomat. Gre pravzaprav za lepšo obliko obsežnega preprostega odzivnega agenta, saj se lahko vsakega agenta s stanji pretvori v SRA (vendar pa bo le-ta pogosto imel globoko gnezdene if-stavke, poleg tega pa bo pogosto pregledoval pogoje, ki jih ni potrebno pregledovati vsakič). Tako ga pogosto delitve v literaturi opuščajo ali pa ga razporedijo kar med preproste odzivne agente, vendar pa ga bomo tu vseeno omenili posebej, saj je ta implementacija izredno pogosta v igrah (tudi pri nadgradnjah, ki jih bomo omenili v nadaljevanju, se osnovna zgradba – avtomat s stanji – v igrah večinoma ohrani), pojavi pa se tudi v splošnem kot osnova arhitekture v scenarijih, kjer je potrebna preprostost in učinkovitost agentov. [31, 13]

V osnovi gre za avtomat s stanji, vsako od stanj pa je sam svoj preprost

odzivni agent. Agent si zapomni svoje notranje stanje in glede na to, v katerem stanju je, izbere pravo odločitveno funkcijo za izbiro akcije glede na trenutno zaznavo. Poleg tega pa mora imeti vnaprej določene tudi pogoje, pod katerimi bo prehajal med stanji. Tako bi lahko rekli, da je agent s stanji pravzaprav preprost odzivni agent, sestavljen iz preprostih odzivnih podagentov.

Ker še vedno za izbiro akcije uporablja le trenutno zaznavo, se prav tako kot SRA lahko slabo odreže v delno vidnih scenarijih, vendar pa je ponavadi zaradi nekoliko večje kompleksnosti njegovo obnašanje bolj zanesljivo in se večinoma neskončnim zankam lahko izogne tudi brez naključnosti.

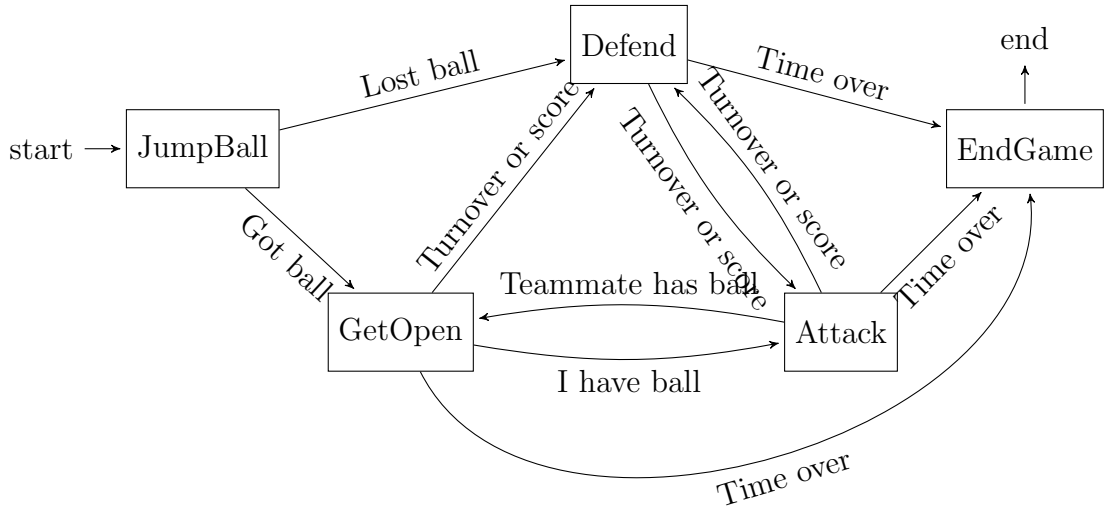
Tako kot SRA se bo tudi agent s stanji slabše odrezal v sekvenčnih scenarijih.

Ta pristop se v igrah uporablja zelo pogosto – tako ga na primer srečamo že v eni prvih 3D iger Tomb Raider (1996), kjer nasprotniki v stanju miru hodijo naokoli po neki okolici, ko pa se jim Lara (oseba, ki jo upravlja igralec) približa, pa preidejo v stanje napadanja, v katerem se premikajo direktno proti Lari in jo poskušajo ugrizniti, udariti ali ustreliti - odvisno od tega, kdo ali kaj ta nasprotnik je.

Prav tako je pristop pogost v športnih igrah (FIFA, NBA, NHL itd.), kjer agenti menjajo stanja glede na to, kdo ima trenutno žogo. Poenostavljeno skico avtomata stanj takega agenta lahko vidimo na sliki 3.2 – v resnici bo seveda imel še nekaj dodatnih stanj za bolj posebne situacije (npr. metanje prostih metov, skoki).

Agenti s stanji so tudi mimoidoči v starejših igrah GTA (do GTA: San Andreas (2004) [5] – kasnejši so že nekoliko kompleksnejši). Sprva mirno hodijo po pločniku, če pa jih igralec napade (ali v nekaterih primerih, če igralca zaznajo), pa preidejo v stanje bežanja ali napadanja.

V igri RPG The Elder Scrolls V: Skyrim (2011) [7] pa agente s stanji predstavljajo skoraj vsi agenti v svetu. Obnašajo se podobno kot tisti, ki smo jih omenili v igri Dragon Age v prejšnjem razdelku, imajo pa dodano še stanje bežanja ali napadanja, v katerega preidejo, če jih igralec napade, jim



Slika 3.2: Preprost SBA za igro NBA. Če ima žogo nasprotna ekipa, pokriva svojega igralca, če ima žogo soigralec, se odkriva, če ima žogo on, pa napada na koš.

kaj ukrade ali pa v njihovi bližini stori kaj nezakonitega.

Tudi večina primerov, ki jih bomo omenili v sledečih razdelkih, kot osnovo uporablja agenta s stanji.

3.4 Agent s spominom

Agent s spominom (angl. *memory-based agent – MBA*) si lahko ob zaznavanju zapomni podatke iz stanja sveta, ki se mu zdijo pomembni in jih nato v kasnejših izbirah akcij uporablja.

Taka nadgradnja je pogosto v pomoč ali celo potrebna v delno vidnih scenarijih (kot smo omenili v prejšnjih dveh razdelkih, se brez nje agenti tam lahko odrežejo zelo slabo) in v sekvenčnih za lažje odločanje na podlagi prejšnjih akcij in stanj sveta.

Prav tako podatke o spremembah stanj lahko uporabimo v neznanih scenarijih (npr. v pomoč pri učenju). V tem primeru agentu pravimo agent z

modelom sveta (angl. *model-based agent*). Model sveta vključuje stanje in posledice agentove akcije na to stanje.

Težave pa bodo agenti s spominom vseeno imeli v stohastičnih scenarijih, saj se lahko deli sveta, ki jih ne zaznavamo, spremenijo in v trenutku odločanja v resnici niso več taki, kot jih ima agent v spominu.

Ta nadgradnja se v igrah seveda uporablja pogosto, a večinoma skupaj z drugimi (predvsem s ciljno usmerjenostjo).

Pogosto jo srečamo na primer pri paznikih, ki igralca lovijo in nasprotnikih v streljaških igrah (oboji si morajo ob izgubi igralca iz vidnega polja zapomniti, kje so ga nazadnje videli, saj bi sicer nemudoma opustili zasledovanje, ker igralca ne zaznavajo več).

V igri Skyrim agente s spominom (in s stanji, kot omenjeno že v prejšnjem razdelku) predstavljajo nekateri bolj napredni NPCji, ki si zapomnijo, kako se je igralec do njih obnašal v preteklosti (npr. ali jim je pomagal ali ne, ko so ga prosili za pomoč). To nato uporabijo pri odločanju, kako se bodo do igralca vedli.

3.5 Ciljno usmerjen agent

Do sedaj smo opisovali agente, ki se obnašajo zelo enolično po vnaprej določenih pravilih. Če bomo takega agenta poskusili uporabiti za enako nalogo z nekoliko drugačnimi parametri, bo pogosto odpovedal. Prav tako lahko odpove, če pride do stanja sveta, ki ga programer ni točno predvidel. To je problematično, če od agenta želimo zmožnost prilagajanja na okolje. Zato agentom dodamo cilje in tako dobimo ciljno usmerjenega agenta (angl. *goal-based agent – GBA*). Ta se o akcijah ne bo več odločal na podlagi vnaprej določenih fiksnih pravil, ki si jih je izmislil programer, pač pa na podlagi tega, katere ga bodo pripeljale do želenega cilja.

Pomembno je omeniti, da agent cilje vedno ima, saj je vsaj en določen že s samo nalogo scenarija. Ciljna usmerjenost pa pomeni to, da agent cilje (ne nujno samo končni cilj naloge) direktno uporablja v izbiri akcije. V ta namen

mora agent imeti zmožnost planiranja, prav tako mora biti sposoben določiti, če ga akcija lahko pripelje do rešitve ali ne. Na nek način torej želimo od agenta napovedovanje prihodnosti – v umetni inteligenci temu problemu bolj formalno pravimo preiskovanje prostora stanj.

Eden najpogostejših problemov, za katerega se taki agenti uporabljajo, je iskanje poti v svetu. Agent se mora na vsakem koraku odločiti, v katero smer bo nadaljeval, da pride do določenega cilja. Pri tem predpostavimo, da pozna stanje sveta (kje so ovire, preko katerih ne more).

Če bi se tega problema lotili brez ciljne usmerjenosti, bi morali agentu podati točna navodila za pot do cilja v stilu "hodi naravnost", "zavij levo" ipd. ali pa uporabiti nekaj naključnega gibanja in upati, da ga bo to pripeljalo do cilja ali do boljše pozicije za nadaljnje premikanje. Če bi nato tega istega agenta poslali na nekoliko premaknjen cilj, bi popolnoma odpovedal. Zato za rešitev tega problema potrebujemo ciljno usmerjenost.

Za reševanje tega problema se najpogosteje uporablja algoritem A*, ki je natančneje opisan v razdelku 5.3.3. Ta je zelo priročen, saj v praksi pot najde hitro in zanesljivo. Poleg tega je pot, ki jo najde, tudi najkrajša možna.

Ciljna usmerjenost omogoča tudi dobro prilagajanje na spremembe v svetu, saj lahko na vsakem koraku spremenimo trenuten plan, če se pojavi boljša alternativa (npr. odprejo se vrata, spusti se most, stražar zaspi ipd.). Zato se taki agenti ponavadi dobro znajdejo tudi v stohastičnih scenarijih, čeprav bolje delujejo v deterministčnih. Bolje od prejšnjih se odrežejo predvsem v sekvenčnih scenarijih zaradi sposobnosti planiranja, možnost prilagajanja pa jim pomaga tudi v delno vidnih scenarijih (čeprav bodo v nekaterih primerih še vedno odpovedali – če na primer ne poznajo sveta in morajo poiskati pot v njem, je to za ciljno usmerjenega agenta z algoritmom A* pogosto nemogoč problem oz. ga moramo reševati z naključnostjo).

Prav zaradi problema iskanja poti je ciljna usmerjenost poleg avtomata s stanji verjetno najpomembnejša nadgradnja agentov v svetu iger. Zato jo v svetovih, kjer je ta problem prisoten, najdemo skoraj vedno.

Tako je na primer prisotna v novejših streljaških igrah, kjer nasprotniki

znajo priti do igralca (ali do točke, kjer so ga nazadnje videli) tudi, če ta pot ni trivialna. Kot primer omenimo npr. The Last of Us (2013) [8], kjer nasprotnike predstavljajo zombiji (oz. okuženi ljudje) in navadni ljudje. Oboji znajo v svetu najti pot do točke, kjer so igralca videli ali slišali.

Najdemo jo tudi v igri Skyrim pri bolj pomembnih osebah, ki morajo potovati po svetu, slediti igralcu, ali ga kam voditi in celo kdaj sami odpreti kakšna vrata, da lahko pridejo do želenega cilja.

3.6 Agent s funkcijo koristnosti

Kot smo omenili v prejšnjem razdelku, algoritem A* (ob nekaterih predpostavkah, ki za nas niso preveč pomembne) vedno najde najkrajšo pot. Vendar pa to z vidika ciljno usmerjenega agenta ni tako pomembno – bolj je pomembno to, da pot vedno najde (če ta obstaja), in sicer v razumnem času – tako mu zagotavlja, da bo dosegel svoj cilj, kar je tudi vse, kar ciljno usmerjen agent zahteva.

Agent s funkcijo koristnosti (angl. *utility-based agent – UBA*) pa želi ne le doseči cilj, temveč to storiti na najboljši možni način. Če gledamo na nalogo kot prostor stanj, v katerem iščemo pot do cilja, imamo lahko v njem več ciljnih vozlišč. Ta vozlišča lahko vsa predstavljajo isti cilj (npr. položaj v svetu, dobljeno igro nogometa ipd.), imajo pa lahko različne lastnosti (npr. zmaga z dvema goloma razlike ali zmaga z enim golom razlike). Za ciljno usmerjenega agenta so v prostoru stanj vsa ciljna vozlišča, ki predstavljajo isti cilj, ekvivalentna – izbere preprosto prvega, ki ga v preiskovanju stanj najde. Agent s funkcijo koristnosti pa se med poljubnima ciljnima vozliščema odloči, katero mu bolj ustreza in se lahko med medsebojno izključujočimi se ciljnimi vozlišči odloči za boljšega (npr. zmaga z dvema goloma razlike mu bo bolj všeč kot zmaga z enim golom – to sta medsebojno izključujoči ciljni vozlišči, saj ne moremo obeh doseči hkrati).

V ta namen uvedemo funkcijo koristnosti (angl. *utility function*), ki predstavlja agentov približek mere uspeha, ki jo bo v nekem ciljnem vozlišču do-

segel. To funkcijo nato poskuša s svojimi akcijami maksimizirati. Ta agent je torej prvi, ki se mere uspeha dejansko zaveda – mera uspeha končno ni le način za naše ocenjevanje delovanja agenta, pač pa jo uporabljamo že pri razvoju in agent z njeno pomočjo izbira akcije.

Tako na primer, če želimo pri iskanju poti nalogo opraviti čim hitreje, za funkcijo koristnosti uporabimo oceno časa, ki jo bomo za pot potrebovali (oz. v primeru konstantne hitrosti preprosteje uporabimo dolžino poti), pomnoženo z -1. Z -1 moramo oceno časa oz. dolžino poti pomnožiti, ker po definiciji večja koristnost pomeni bolj uspešno delovanje.

Tako dobimo najhitrejšo od poti, ki nas vodijo do cilja. Vendar pa nam, kot smo to že omenili, ta problem reši tudi že ciljno usmerjen agent, saj nam algoritem A*, ki je zanj najprimernejši, zagotavlja tudi najkrajšo pot. Podobno se zgodi tudi pri mnogih drugih problemih iskanja ciljev, saj najpreprostejšo (najcenejšo) pot do cilja pogosto tudi najlažje najdemo.

Velika razlika med ciljno usmerjenim agentom in agentom s funkcijo koristnosti pa se pokaže v delno vidnih in stohastičnih scenarijih. Tako se na primer pri problemu, ki smo ga omenili v prejšnjem razdelku, ko ne poznamo sveta, v katerem iščemo pot, agent s funkcijo koristnosti odreže mnogo bolje. Iz trenutno vidnih položajev (vmesnih ciljev) si bo izbral nov cilj tako, da bo maksimiziral verjetnost, da od tam čim hitreje doseže želen končni cilj.

Poleg tega, kadar ima na voljo veliko ciljev (točk interesa), ki jih želi vse obiskati, zna najti dobro pot med njimi. Če pa se cilji med seboj izključujejo, pa zna med njimi izbrati najboljšo kombinacijo ciljev, ki jo lahko doseže.[1] Glavna prednost agenta s funkcijo koristnosti pred ciljno usmerjenim je torej v splošnem sposobnost **izbire najboljših ciljev**.

Tako smo prišli že zelo blizu človeškemu obnašanju. Če na primer pogledamo igralca, ki prvič igra kako igro z odprtim svetom, v kateri postavitve

[1]Cilji niso nujno fizične lokacije za iskanje poti. Lahko gre tudi za bolj abstraktne pojme, na primer pomagati prijatelju in premagati sovražnika – to sta pogosto cilja, ki ju ne moremo opraviti hkrati, zato se mora agent med njima stalno odločati. Do podobnih situacij pogosto pride v večagentnih scenarijih, kjer prav zato taki agenti delujejo zelo dobro. [25]

sveta še ne pozna, bo pogosto izbiral cilje, za katere verjame, da ga vodijo proti želenemu položaju. Poleg tega v sobah z opremo (npr. municijo, orožjem, baterijami) pogosto uporabi enega najhitrejših sprehodov, tako da pobere vse predmete, ki so mu na voljo. Razmeroma dobro pa se ljudje večinoma odrežemo tudi pri odločitvah med nasprotujočimi si cilji. V naslednjem razdelku pa bomo dodali še eno lastnost, ki bo agente pripeljala do prav zares že skoraj človeškega obnašanja.

V igrah agente s funkcijo koristnosti najdemo bolj redko, saj so pogosto preveč kompleksni za scenarije, ki se v igrah pojavljajo (da nimamo pretirane kompleksnosti, se v igrah pogosto pri iskanju poti in podobnih problemih na primer uporabi popolna vidnost scenarija, tudi če ga drugače imamo za delno vidnega, zato ni potrebe po agentih s funkcijo koristnosti). V igrah jih najdemo predvsem tam, kjer res potrebujemo občutek človeškega odločanja.

Primer so sobojevniki iz RPGjev iz serij Dragon Age in Final Fantasy, ki spremljajo igralca (igralec lahko kontrolira poljubni lik v skupini, preostale kontrolira računalnik). Ti se morajo v bitki konstantno odločati med nasprotujočimi si cilji – naj napadejo nasprotnika ali naj zdravijo sobojevnika? Naj se branijo pred sovražnikovim napadom, naj ga ignorirajo ali pa naj zbežijo na varno razdaljo? Pri nekaterih odločitvah jim sicer lahko igralec pomaga z vnaprej določenimi pravili, večinoma pa se morajo o njih odločati sami (odvisno tudi od igre – v nekaterih imajo več svobode, v nekaterih manj). V Skyrimu pri sobojevniku, ki igralca spremlja, takih odločitev med cilji ni videti – obnaša se ciljno usmerjeno.

Agente s funkcijo koristnosti najdemo tudi v novejših realno-časovnih strateških igrah (angl. *real-time strategy – RTS*), kjer je scenarij delno viden in mora agent ugibati, kaj igralec (in preostali agenti) trenutno počnejo. Poleg tega je iskanje poti v teh problem, v katerem najboljša pot ni vedno najkrajša, zato so potrebni drugačni pristopi. [22] V starejših ima agent preprosto dostop do celotnega stanja sveta in je poštenost dosežena na drugačne načine.

Prav tako jih najdemo tudi v igrah, kjer dejansko nadomestijo prave

človeške igralce, torej večigralskih igrah, kot je na primer Counter-Strike: Global Offensive (2012) [1]. V tej igri morajo t.i. boti (računalniško vodeni nadomestki sicer pravih igralcev) stalno sprejemati strateške odločitve – naj branim točko ali grem iskat nasprotnika? Naj pomagam soborcu, ki je napaden, ali nadaljujem z misijo? To je odvisno od tega, kdo ima bombo (povezano z misijo), kje so nasprotniki, kaj od njega želi igralec (ki z boti lahko komunicira preko radijskih ukazov) itd. Poleg tega pa morajo odločitve sprejemati tudi med dvobojem z nasprotnikom – naj stojim pri miru in streljam, naj tečem naokoli in streljam, ali naj se skrijem? To je odvisno od njihovega orožja, od orožja nasprotnika in od tega, koliko nabojev imajo še na voljo.

Nazadnje pa lahko (nekoliko prilagojene in specializirane) agente s funkcijo koristnosti najdemo med šahovskimi agenti, saj so od do sedaj naštetih agentov za igranje šaha daleč najbolj primerni.

3.7 Učljiv agent

Učljiv agent (angl. *learning agent – LA*) je najbolj kompleksen izmed agentov v tej delitvi. Kot edini od naštetih agentov se dobro znajde tudi v nepoznanih scenarijih, prav zaradi zmožnosti učenja. V sklopu te naloge ga sicer ne bomo uporabljali (v svetu igre potrebujemo namreč preverjeno in zanesljivo obnašanje agentov, učljivi agenti pa temeljijo na podatkovnem rudarjenju in strojnem učenju, torej je zanesljivost skoraj nemogoče zagotoviti, njihovo testiranje pa je prav tako dolgotrajno in dokaj nezanesljivo), vseeno pa ga tu predstavimo in na kratko opišemo, saj je pomemben del delitve.

Učljiv agent je razdeljen na dva dela – eden je zadolžen za izbiranje akcij (to je bila v dosedanjih funkcija celotnega agenta), drugi pa ocenjuje, kako dobro se je agent zaenkrat v scenariju obnesel in predlaga izboljšave obnašanja. Zaradi predlogov drugega elementa bo prvi pogosto izbral akcijo, ki je sicer ni ocenil kot najboljšo, vendar pa lahko zaradi informacij, ki jih bo od nje izvedel, v prihodnosti močno izboljša svoje obnašanje.

Tak agent bo pogosto v preprostih scenarijih prikazal slabše rezultate kot prejšnji, saj bo po nepotrebnem poskušal izboljšati svoje delovanje in bo scenarija že konec, ko bo prišel do tako dobrega, kot ga prikažejo prejšnji. Bolje se bo odrezal šele v zelo kompleksnih scenarijih, največja razlika pa se seveda pokaže v nepoznanih scenarijih. Zelo pomembna lastnost, ki mora veljati za dobro delovanje učljivega agenta, pa je dolg čas življenja v scenariju, saj potrebuje dovolj časa, da svoje vedenje optimizira.

V igrah so taki agenti zelo redki, saj zaradi kompleksnosti niso primerni za igre z veliko agenti, poleg tega pa se pogosto obnašajo slabše kot preprostejši v scenarijih, ki jih najdemo v igrah. Večinoma se pojavljajo v bolj eksperimentalni obliki v akademskih raziskavah [23, 29, 14, 35], v komercialnih igrah pa jih ne najdemo veliko. Najdemo pa jih med šahovskimi agenti, kjer je sposobnost učenja zelo zaželena. Taki agenti seveda nadaljujejo svoje učenje in delovanje skozi več iger – ne začnejo ob začetku vsake igre znova. Pogosto se učijo tudi iz že odigranih iger iz zgodovine, ki so jim na voljo v kakšnih podatkovnih bazah.

Učljive agente bi lahko uporabili tudi v igrah RTS, kjer igralci lahko čez nekaj časa postanejo predvidljivi in bi se agent z zmožnostjo učenja tedaj lahko vnaprej pripravil na njihove napade.

3.8 Alternativni pristopi k arhitekturi agentov

Kot omenjeno v začetku tega poglavja, naša delitev ni edina v uporabi. Obstajajo tudi drugi pristopi k arhitekturi agentov, razviti za drugačne potrebe.

Arhitektura BDI (angl. *belief-desire-intention*) je bila razvita za logične programske jezike in sledi Bratmanovemu modelu človeškega razmišljanja [17]. Če jo poskusimo opisati z našimi izrazi, prepričanje (angl. *belief*) predstavlja agentov model sveta, želja (angl. *desire*) predstavlja cilj, namen (angl. *intention*) pa predstavlja izbiro akcije. Tak agent sicer spominja na ciljno usmerjenega, kot smo ga definirali mi, vendar pa se bolj kot na implemen-

tacijo iskanja poti do cilja osredotoča na to, kako najbolje predstaviti te tri pojme – za preiskovanje pa potem poskrbi že programski jezik sam.

Ta arhitektura je danes zelo pogosta. Tako se na primer uporablja celo v vojaških sistemih kot alternativa (lahko tudi dopolnitev) prej bolj popularni arhitekturi z zanko zaznaj-procesiraj-odloči se-izvedi akcijo (angl. *observe-orient-decide-act – OODA loop*). [19]

Conte [18] že na višjem nivoju deli agente drugače, in sicer loči med racionalnimi in inteligentnimi: racionalni skupaj s ciljno upravljanimi (angl. *goal-directed*) predstavljajo poseben primer inteligentnih, vsi skupaj pa spadajo med ciljno orientirane (angl. *goal-oriented*). Tako torej pove, kot smo to omenili mi, da imajo vsi agenti določene cilje (tudi, če agentu niso poznani), inteligentne agente definira podobno kot smo jih mi v razdelku 1.1, racionalni in ciljno upravljani pa predstavljajo nekaj podobnega našim definicijam agentov s funkcijo koristnosti in ciljno usmerjenih agentov.

Poleg tega imamo še hibridne arhitekture, ki združujejo več pristopov. Løland [26] na primer omenja arhitekturi InteRRaP in TouringMachines in se nato posveti možnosti uporabe takih hibridnih pristopov v svetu iger (bolj natančno v svetu streljaških iger iz serije Quake).

TouringMachines uporablja tri plasti in kontrolni podsistem, ki odloča o tem, katera od plasti bo v določenem trenutku imela kontrolo nad agentom. Odzivna plast (angl. *reactive layer*) deluje kot preprost odzivni agent. Načrtujoča plast (angl. *planning layer*) deluje podobno ciljno usmerjenemu agentu, le da uporablja namesto preiskovanja vnaprej pripravljene sheme (angl. *schemas*). Plast modeliranja (angl. *modelling layer*) pa skrbi za vzdrževanje modela sveta ter pripravlja in posreduje cilje načrtujoči plasti.

InteRRaP ima prav tako tri plasti. Spodnja je vedenjska plast (angl. *behaviour layer*), ki deluje kot preprost odzivni agent. Srednja je načrtujoča plast, ki prav tako kot v prejšnji arhitekturi rešuje lokalne probleme s pregledovanjem vnaprej določenih shem. Vrhnja pa je sodelovalna plast (angl. *cooperation layer*), ki deluje nekoliko bolj podobno agentu s funkcijo koristnosti, saj išče cilje, ki bodo najboljši ne le za agenta samega, pač pa tudi

za preostale agente, ki jim poskuša pomagati. V vsakem koraku primerno akcijo najprej poskuša določiti vedenjska plast. Če ji to ne uspe, preda kontrolo načrtujoči, ta pa (če ji ne uspe najti rešitve) sodelovalni. Nato plast, ki ima kontrolo izvede akcijo, ki jo vodi do rešitve (pri tem si lahko pomaga tudi z nižjimi plastmi).

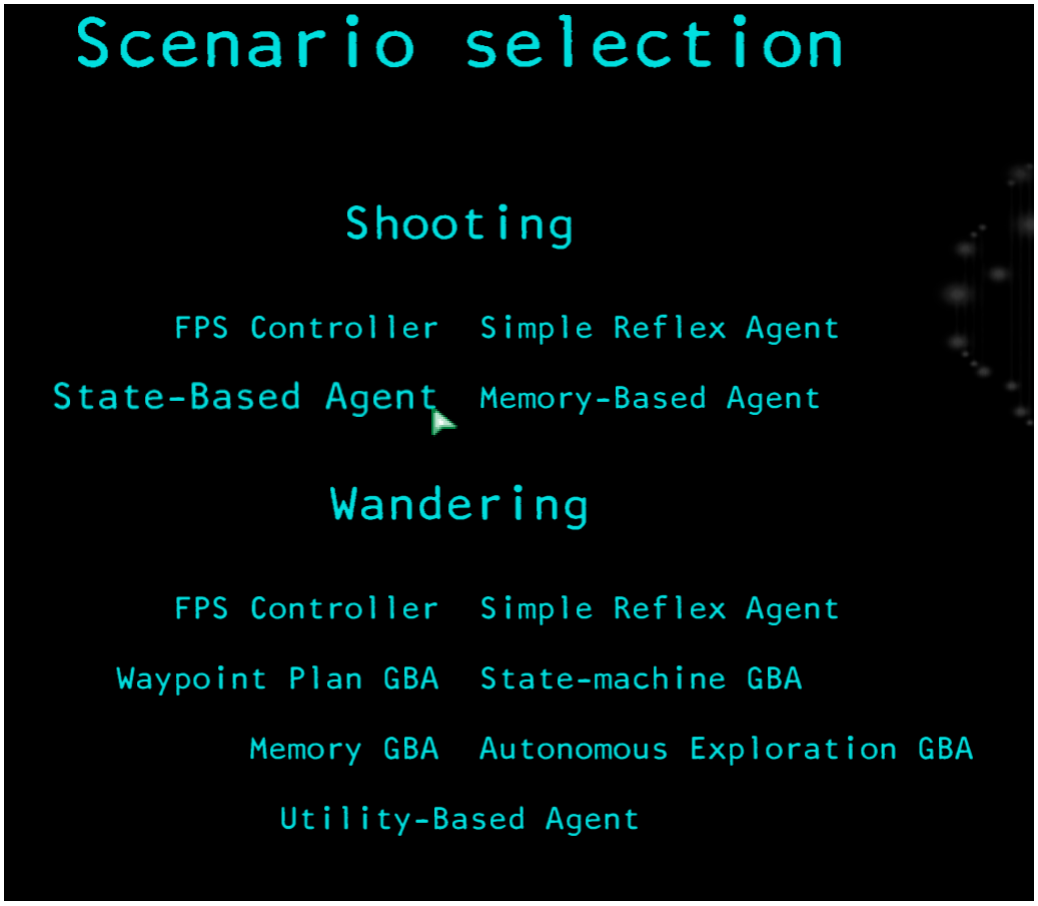
Poglavje 4

Testna scenarija

Spoznali smo vse agente, ki jih bomo med seboj primerjali, sedaj pa potrebujemo za to primerjavo primeren scenarij. Navadno sta okolje in naloga definirana že preden sploh začnemo razmišljati o agentih, ker je naš cilj prav primerjava agentov, pa bomo tokrat scenarij načrtovali tako, da bo čim bolje prikazal dobre in slabe lastnosti agentov.

V ta namen bomo uporabili dva scenarija. Ciljna usmerjenost namreč prinese tako ogromno spremembo v delovanju agentov, da je težko narediti en sam scenarij, ki bi ohranil potrebo po vsaki od nadgradenj, ki smo jih našteli v prejšnjem poglavju, vseeno pa bi v njem vsi agenti delovali vsaj približno inteligentno. Ciljno usmerjeni agenti se namreč uporabljajo za reševanje problemov, pri katerih agenti brez ciljne usmerjenosti večinoma povsem odpovejo.

Zato primerjavo raje izvedemo v dveh scenarijih in tako pri vsakem agentu veliko bolje prikažemo njegove zmožnosti in slabosti. Vsakega od scenarijev bomo točno definirali in opisali njegove lastnosti, ki jih bomo v 5. poglavju nato uporabili pri izbiri agentov za implementacijo. Obenem bomo opisali še uporabniške kontrole, ki so uporabniku v pomoč v aplikaciji.

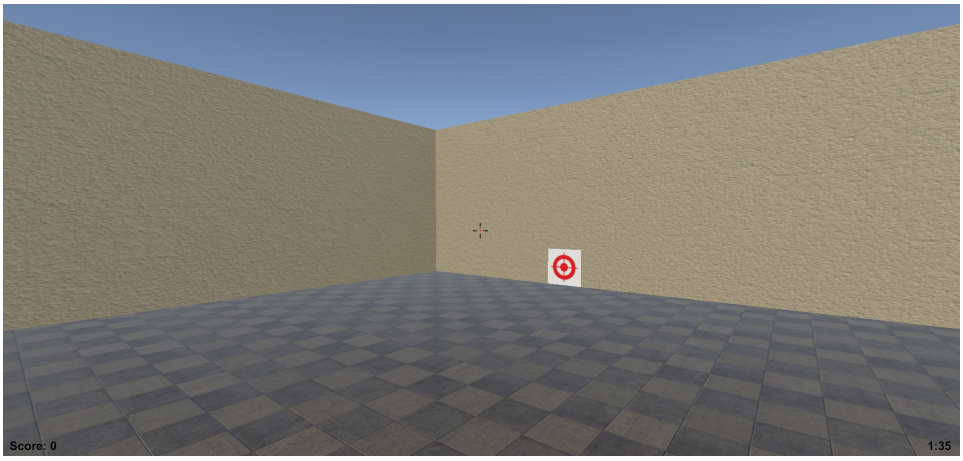


Slika 4.1: Glavni meni aplikacije.

4.1 Glavni meni

Ko našo aplikacijo zaženemo, najprej pridemo do izbire scenarija in agenta, kot vidimo na sliki 4.1. V zgornjem delu so agenti prvega scenarija, spodaj pa agenti drugega. Za izbiro preprosto z miško kliknemo na želenega agenta. Ob izboru nam nekateri agenti v drugem scenariju omogočajo še izbiro premaknjenega scenarija – potreba po tem bo jasna kasneje.

Prav tako imamo pri obeh scenarijih na voljo še možnost, da sami kontroliramo lik v obliki prvoosebnega krmilnika (angl. *First Person Controller*).



Slika 4.2: Okolje v prvem scenariju.

4.2 Prvi scenarij

4.2.1 Opis scenarija

Prvi scenarij je preprost primer streljanja tarč iz sveta prvoosebnih streljaških iger. Opišimo ga z opisom PEAS, ki smo ga spoznali v 2. poglavju. Najlažje je začeti z opisom okolja in naloge, ki jo mora agent opravljati. Kot del tega opišemo tudi agentovo predznanje o okolju.

Okolje je majhna soba, v kateri se agent nahaja. Na eni od sten te sobe se nahaja tarča, ki jo mora agent zadeti. Ko tarčo zadane ali pa preteče 30 sekund od pojavitve, tarča izgine in pojavi se naslednja. Tarče so štiri in se v teku scenarija stalno pojavljajo v enakem vrstnem redu (ko izgine zadnja, se pojavi spet prva). Ta vrstni red je naključno določen ob začetku scenarija.

Agent vnaprej ne pozna njihovih položajev, ne ve, koliko jih je, in ne ve, v kakšnem vrstnem redu se bodo pojavile. Zaveda pa se, da so tako položaji tarč kot tudi vrstni red vnaprej določeni. Na ta način lahko dobro prikažemo vrednost spomina o stanju sveta, saj si bo agent z njim lahko močno pomagal (vseeno pa mu spomin ni nujno potreben).

Scenarij se konča po 2 minutah. V spodnjem desnem kotu zaslona lahko stalno spremljamo, koliko časa še ostaja. Okolje lahko vidimo na sliki 4.2.

Mera uspeha	**Okolje**	**Akcije**	**Senzorji**
Število zadetih tarč	Soba s tarčami na stenah	Premikanje kamere, streljanje	Vid, sluh, ura

Tabela 4.1: Opis PEAS za prvi testni scenarij.

Za mero uspeha uporabimo število točk, ki jih agent zbere (za vsako zadeto tarčo dobi eno točko). Ker je čas scenarija vnaprej določen in konstanten, na tak način dobimo preprosto in zanesljivo mero uspeha. Število točk lahko na zaslonu spremljamo v spodnjem levem kotu, izpiše pa se nam tudi ob koncu scenarija.

Akcije, ki so agentu na voljo, so obračanje kamere (simuliramo igralčevo premikanje miške) in streljanje naravnost v smeri pogleda (streli sicer niso popolnoma natančni, lahko zgrešijo za nekaj stopinj). Ob akciji streljanja kamera rahlo trzne navzgor, saj orožje deluje na igralca s silo sunka strela (angl. *recoil*). Poleg tega je frekvenca strelov (angl. *rate of fire*) omejena. Obračanje kamere agentu pomaga pri iskanju tarče in merjenju, streljanje pa uporabi, da lahko tarčo dejansko zadane.

Glavni senzor, ki je agentu na voljo, je vid. Kot že rečeno, zanj uporabimo programsko implementirani približek, saj ima sicer računalnik na voljo informacijo o celotnem svetu. Natančnejši opis implementacije je v razdelku 5.1. Poleg tega pa agent ob streljanju sliši, ali je tarčo zadel ali ne, na voljo pa ima tudi uro (torej dostop do preteklega časa v scenariju).

Zgoščen povzetek opisa je podan v tabeli 4.1.

4.2.2 Lastnosti scenarija

Sedaj si poglejmo še lastnosti scenarija. Najprej jih spet predstavimo kar v tabelni obliki (tabela 4.2).

Vendar pa tak enostaven opis lastnosti ni dovolj za izbiro primernih agentov, saj moramo o vsaki lastnosti nekoliko bolje premisliti.

Scenarij je delno viden, saj v vsakem trenutku vidimo le del sobe, nimamo

Vidnost	Št. ag.	Determ.	Epiz.	Stat.	Diskr.
Delno viden	Enoagenten	Stohastičen	Sekvenčen	Dinamičen	Zvezen

Tabela 4.2: Lastnosti prvega testnega scenarija.

torej pregleda nad celotno sobo.

Stohastičen sicer je, vendar pa se nepredvidljiva sprememba zgodi samo v primeru, ko 30 sekund ne zadanemo tarče, torej je v splošnem verjetnost, da se stanje spremeni nepovezano z akcijo, zelo majhna. Zato ga lahko obravnavamo kot skoraj deterministočnega in nam o stohastičnih spremembah ni potrebno razmišljati.

Podobno kot za stohastičnost velja tudi za dinamičnost – stanje se sicer lahko spremeni med razmišljanjem, vendar pa se bo to zgodilo tako poredko, da nam o tem ni treba skrbeti in lahko scenarij obravnavamo kot statičen.

Scenarij je sekvenčen, saj akcije vplivajo na prihodnje izbire – če obračamo kamero v pravo smer, bomo tarčo našli, sicer pa ne, prav tako pa bomo tarčo zgrešili, če pred tem ne namerimo vanjo. Vendar pa je ta povezanost akcij dovolj preprosta, da lahko kompleksnejše načrtovanje nadomestimo s preprostimi vgrajenimi pravili, zato tudi zmožnosti načrtovanja agent ne potrebuje.

Stanja sveta so zvezna (koordinate tarč so namreč idejno trojčki realnih števil), tako tudi akcije (smer obračanja kamere predstavljajo dvojčki realnih števil) in zaznave. Prav tako je zvezen tudi potek časa. Tako bomo za stanja sveta, akcije in zaznave uporabljali približke v plavajoči vejici, čas pa bo v implementaciji tekel v korakih s frekvenco slik (angl. *frame rate*).

Edina problematična lastnost tega scenarija je torej delna vidnost, ostale lastnosti pa so ugodne ali zanemarljivo neugodne. Ta scenarij bo dobro prikazal delovanje preprostejših agentov z deterministočnimi vnaprej predpisanimi pravili, poleg tega pa bo prikazal, kako lahko spomin vpliva na obnašanje v delno vidnih okoljih (čeprav bosta tudi agenta brez spomina nalogo do neke mere opravila).

4.2.3 Uporabniške kontrole v aplikaciji

Uporabnik v aplikaciji nima posebnih možnosti interakcije, lahko samo opazuje delovanje agenta. S tipko `Esc` lahko trenuten scenarij prekine in se vrne na glavni meni, ob koncu scenarija in izpisu števila točk pa to stori s pritiskom na katerikoli gumb.

V prvoosebnem krmilniku se lahko uporabnik obrača z miško, s pritiskom na levi gumb miške (ali tipko `Ctrl`) strelja, omogočeno pa mu je tudi premikanje s tipkami `WASD`.

4.3 Drugi scenarij

4.3.1 Opis scenarija

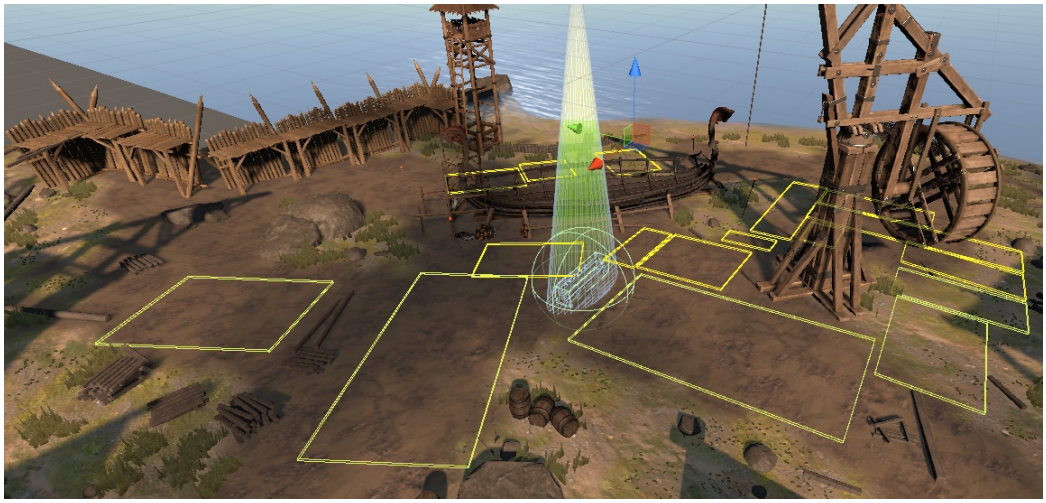
Drugi scenarij je bolj kompleksen in vsebuje med drugim tudi problem iskanja poti, ki smo ga omenili kot osnovni problem v ciljni usmerjenosti. Spet se lotimo najprej opisa okolja in naloge.

Celotno okolje je sestavljeno iz prosto dostopnega primera sveta Viking Village ter stojal in orodij, vse to pa je na voljo na Unityjevi spletni trgovini (angl. *Asset Store*) [10]. Gre za vikinško vas s hišami in čolni, med katerimi so skrita tri stojala za orodja (angl. *tool racks*). Položaji teh stojal so agentu znani ob zagonu scenarija, prav tako pozna tudi postavitev (angl. *layout*) sveta. Vsako od stojal ima z množico kvadrov definirano okolico (gl. sliko 4.3).

Ko agent pride do enega od stojal, se v prej omenjeni okolici tega stojala na petih naključno izbranih položajih pojavijo orodja (sekire). Agent mora te sekire najti in jih vrniti na stojalo. Nato ponovi postopek še pri ostalih dveh stojalih.

Za lažje spremljanje uporabnik nad stojali in sekirami vidi snope rumenkaste svetlobe, če so ti trenutno pomembni za agenta, agent pa snopa ne zazna (z drugimi besedami, agent v implementaciji vida teh snopov ne upošteva).

Scenarij se konča po 7 minutah, lahko pa ga konča tudi agent, če se odloči,



Slika 4.3: Izbrano stojalo in njegova okolica, definirana z (rumenimi) kvadri.

da je s svojim delom zaključil. Pretekli čas lahko spremljamo v zgornjem desnem kotu zaslona.

Kot mero uspeha uporabimo spet število točk, ki tokrat predstavlja število vrnjenih sekir. To lahko med potekom scenarija spremljamo v zgornjem levem kotu, ob koncu pa se nam skupaj s časom tudi izpiše. Ker bo s tako mero uspeha prišlo do enakih ocen nekaterih agentov, uporabimo še sekundarni kriterij, in sicer čas, pomnožen z -1 (večja mera uspeha pomeni boljše delovanje, daljši čas pa slabše delovanje, zato ga pomnožimo z -1).

V primerjavi lahko preprosto najprej primerjamo števila točk, ki jih agenti zberejo, če pa so ta kdaj enaka pa primerjamo še čas. Če pa hočemo imeti formalno definirano mero uspeha kot eno (čim bolj preprosto) funkcijo, pa lahko številu točk preprosto dodamo primerno utež. Tako na primer mero uspeha definiramo kot $f = 10 \times št.\, točk - porabljen\ čas\ v\ minutah$. S tako utežjo bo število točk vedno prevladalo, saj je največji možen čas 7 minut.

Akcije, ki so agentu na voljo, so obračanje in premikanje naravnost v smeri, kamor je trenutno obrnjen. Nanj seveda deluje tudi gravitacija in sila tal, tako da se bo hkrati premikal tudi gor in dol po klancih.

Kot senzor ima, podobno kot v prejšnjem scenariju, spet implementiran vid, poleg tega pa se zaveda tudi svojega položaja v svetu (postavitev sveta

Mera uspeha	Okolje	Akcije	Senzorji
Št. vrnjenih sekir, če je to enako pa še porabljen čas	Vikinška vas: hiše, čolni, poti, mlin ipd.	Obračanje, hoja naravnost	Vid, GPS, ura

Tabela 4.3: Opis PEAS za drugi testni scenarij.

Vidnost	Št. ag.	Determ.	Epiz.	Stat.	Diskr.
Delno viden	Enoagenten	Stohastičen	Sekvenčen	Dinamičen	Zvezen

Tabela 4.4: Lastnosti drugega testnega scenarija.

pa je, kot smo omenili zgoraj, vključena v agentovo predznanje). Spet ima na voljo tudi uro.

Za konec spet povzamemo opis v tabeli 4.3.

4.3.2 Lastnosti scenarija

V tabeli 4.4 imamo seznam lastnosti, ki veljajo za drugi scenarij.

Tabela je enaka tabeli 4.2 iz prvega scenarija. Vseeno pa je scenarij veliko bolj kompleksen in tu vidimo, zakaj ni dovolj le našteti lastnosti. Niso namreč vse črno-belo določene – v prejšnjem scenariju smo večino neugodnih lahko zanemarili.

Scenarij je spet delno viden, razlika od prejšnjega pa je, da tokrat vidimo veliko manjši del sveta – svet ni le večji, pač pa ima tudi mnogo ovir, ki nam pogled zastirajo. Ta lastnost je predstavljala problem že v prejšnjem scenariju, torej bo tokrat kvečjemu še bolj neugodna.

Tokrat ima stohastičnost veliko večji vpliv, saj močno vpliva na zelo pomemben del scenarija – položaji sekir so namreč izbrani naključno. Tako bo veliko večji pomen imela agentova zmožnost preiskovanja, ki jo bomo dobili s ciljno usmerjenostjo.

Za dinamičnost velja podobno kot v prejšnjem scenariju. Edina sprememba, ki se lahko zgodi v svetu med agentovim razmišljanjem, je vpliv gravitacije na sekire ob njihovi pojavitvi, kar pa je nepomembno (večinoma je ta vpliv zelo kratek in majhen). To lastnost torej še vedno lahko zanemarimo in scenarij obravnavamo kot statičen.

Veliko večjo zahtevnost pa bo tokrat povzročila sekvenčnost. Ker je tokrat kriterij tudi čas, bo pomembno, v kakšnem zaporedju obiskujemo cilje. Še bolj pa se sekvenčnost pokaže v sedaj že neslavnem problemu iskanja poti – ta vedno predstavlja sekvenčnost, saj vsak premik vpliva na nadaljnje iskanje poti. Zaradi tega bomo v tem scenariju nujno potrebovali ciljno usmerjenost vsaj na nivoju premikanja med točkami. Brez nje se agent, kot bomo videli, obnese zelo slabo.

Stanja sveta, akcije, zaznave in čas so, podobno kot v prejšnjem scenariju, vsi zvezni, kar pa za izbiro agenta ni tako pomembno.

Neugodnih lastnosti je tokrat torej več. Imamo delno viden, stohastičen, sekvenčen scenarij. Zato bomo že takoj začeli z agentom s ciljno usmerjenostjo na nivoju premikanja med točkami.

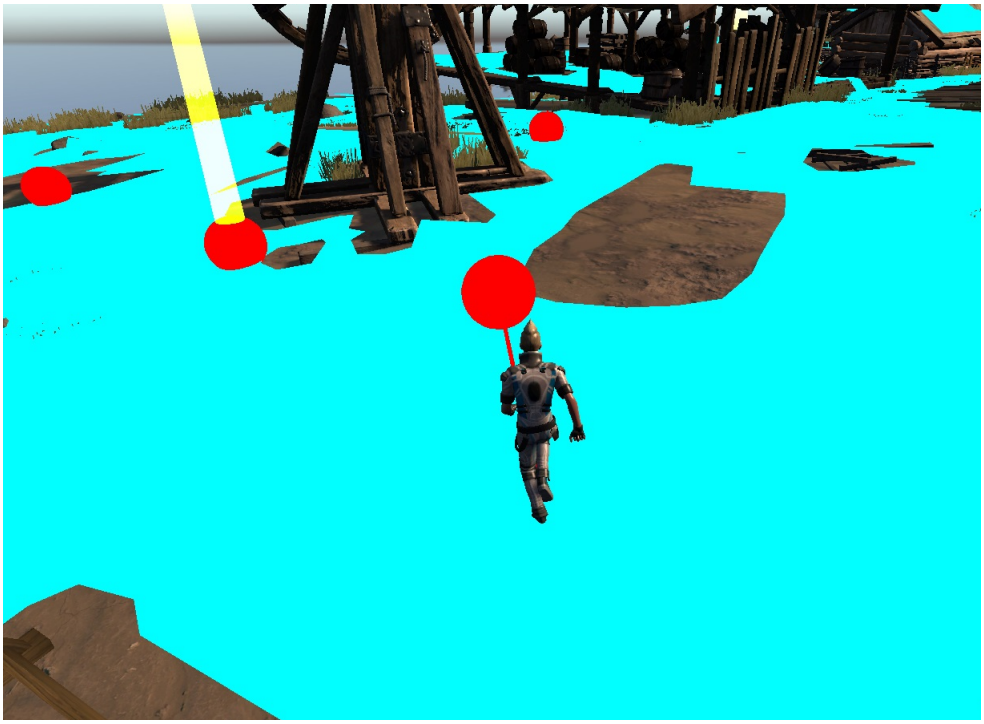
4.3.3 Uporabniške kontrole v aplikaciji

Uporabnik ima enake kontrole kot v prvem scenariju (glej razdelek 4.2.3), poleg tega pa ima omogočenih še nekaj:

- Z levim gumbom miške lahko spremeni kamero. Na voljo so mu tri kamere: prvoosebna (ki se uporablja tudi pri implementaciji vida – gl. razdelek 5.1), tretjeosebna čez rame (angl. *over-the-shoulder – OTS 3rd person*) in pa izometrična. Več o kamerah bomo povedali v razdelku 5.3.1.

- S premikanjem miške levo in desno lahko vrti kamero, vendar le v izometričnem pogledu.

- S pomikanjem koleščka na miški (angl. *scroll wheel*) lahko približuje in



Slika 4.4: Način izrisa navigacije.

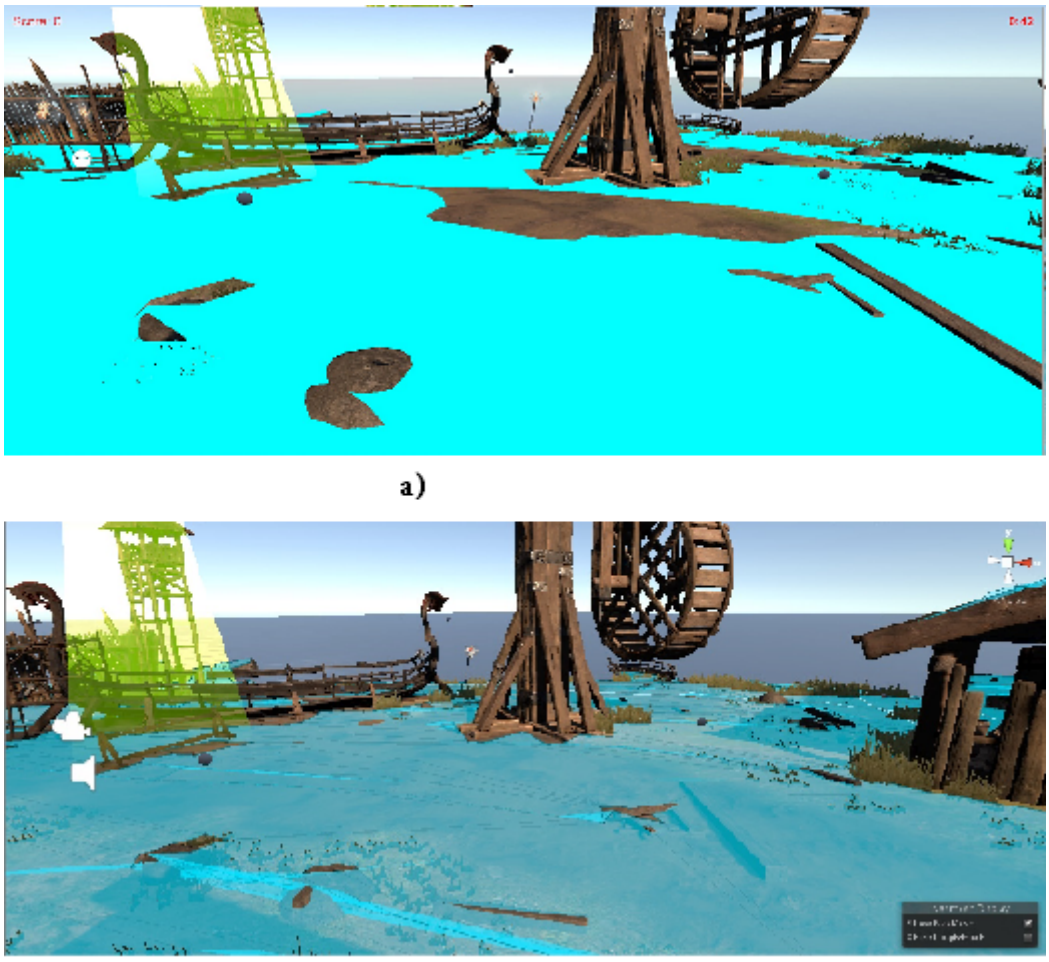
oddaljuje kamero od našega lika, vendar le v izometričnem ali tretjeosebnem pogledu.

- S tipko `N` lahko vključi in izključi način izrisa navigacije, v katerem se mu izrišejo NavMesh, ki predstavlja prehodne površine okolja (več o tej strukturi v razdelku 5.3.3), trenutna izračunana pot po algoritmu A* in ciljne točke (angl. *waypoints*), ki jih trenutno pozna. NavMesh je obarvan svetlo modro, pot je predstavljena z rdečo črto po tleh, točke pa so rdeče krogle, kot vidimo na sliki 4.4.

  Žal izračun triangulacije NavMesha v Unityju ni popolnoma natančen, zato ponekod pride do odstopanj (kot vidimo na sliki 4.5), saj je bilo potrebno pozicijo NavMesha popravljati ročno. Poleg tega je izris črt v Unityju presenetljivo zahteven, zato NavMesh izrisujemo brez robov in se tako na njem ne vidijo trikotniki. Ta dva problema rešimo, če si NavMesh raje ogledamo v Unity Editorju (v meniju Window izberemo

Navigation in nato obkljukamo Show NavMesh in po želji še Show HeightMesh).

V prvoosebnem krmilniku se spet lahko premikamo s tipkami `WASD` in obračamo z miško, levi gumb miške pa ne stori več ničesar.



a)

b)

Slika 4.5: a) Izris NavMesha v pognani aplikaciji v svetlo modri barvi.
b) Izris dejanskega NavMesha v Unity Editorju.

Poglavje 5

Implementacija agentov

V obeh scenarijih bomo začeli z zelo preprostim odzivnim agentom, ki ga bomo nato nadgrajevali do bolj kompleksnih. Preden pa se lotimo implementacije agentov, pa v obeh scenarijih najprej potrebujemo implementacijo najpomembnejšega senzorja – senzorja za vid.

5.1 Implementacija vida

Simulacija vida je problem, ki se v igrah pojavlja zelo pogosto. Ker ga velikokrat uporabljajo tudi najpreprostejši agenti v velikih svetovih (le-teh je ponavadi veliko), mora biti implementiran zelo učinkovito. [24]

V starejših igrah je bil vid pogosto nadomeščen kar s polmerom zaznave ali pa celo s t.i. sprožilcem zaznave (angl. *trigger*), ki je agentovo zaznavo sprožil, ko je osebek prestopil nek vnaprej določen prag. Tako so agenti ostale osebke zaznali takoj, ko so se jim dovolj približali, tudi če v resnici niso bili v njihovem vidnem polju (in niso povzročali hrupa).

V članku Kuiperja in Wenksterna [24] poleg opisa osnovnih metod najdemo številne optimizacije sodobnih algoritmov, mi pa se bomo zadovoljili z implementacijo, ki bo vidnost predmetov preverjala v treh korakih:

1. **Preverjanje oddaljenosti**: Predmetov ne vidimo, če so predaleč stran, zato agentu omejimo vid na neko vnaprej določeno razdaljo.

Izračun te mejne razdalje bi lahko vključili v implementacijo in jo popravljali glede na velikost predmeta, katerega vidnost testiramo (večje predmete vidimo na večjih razdaljah). Vendar pa v naših scenarijih to ni potrebno, saj v prvem preverjamo le vidnost tarč (te so vedno dovolj blizu), v drugem pa le vidnost sekir (ki so vse enake velikosti in imajo zato tudi enako mejo). Zato mejo preprosto nastavimo v vsakem od scenarijev kot konstanto.

Če je predmet dovolj blizu, nadaljujemo postopek, sicer pa predmet ni viden.

2. **Preverjanje kota**: Predmetov ne vidimo, če so izven našega vidnega polja (angl. *field of view – FOV*).

   Zato izračunamo horizontalni in vertikalni kot med vektorjem v smeri agentovega trenutnega pogleda in vektorjem, ki kaže od pozicije agentove kamere pogleda (angl. *vision camera*) proti predmetu. Za kamero pogleda v obeh scenarijih uporabljamo kar prvoosebno kamero, ki jo ima agent.

   Nato primerjamo dobljeni horizontalni kot s horizontalnim kotom vidnega polja kamere in če je večji, predmeta agent ne vidi. Podobno ponovimo tudi z vertikalnim kotom.[1] Pri tem moramo kota vidnega polja kamere še razpoloviti, saj predstavljata razpon celotnega vidnega polja, nas pa zanima kotna oddaljenost od njegove sredine.

   Če sta oba kota v mejah podanih z vidnim poljem kamere, potem nadaljujemo s tretjim korakom, sicer pa postopek prekinemo, saj predmet ni viden.

[1] Pogosto se uporablja tudi preprostejši pristop stožca vidnosti (angl. *vision cone*) namesto piramide vidnosti, ki smo jo za opis vidnega polja uporabili mi. Pri stožcu primerjamo le en kot, poleg tega se izognemo projekcijam in normalizacijam, ki lahko v redkih primerih povzročijo numerično nestabilnost in posledično nenatančne izračune. Vendar pa naš pristop bolje oceni vidno polje, kot ga imamo v igrah – ekran je namreč pravokoten, ne okrogel.

3. **Preverjanje dostopnosti**: Predmetov ne vidimo, če nam kakšen objekt v svetu zastira pogled na njih.

   Zato izvedemo met žarka (angl. *raycast*) od položaja kamere pogleda proti položaju predmeta. Če žarek na poti do predmeta ne trči v noben objekt, je pot prosta in predmet vidimo. Sicer postopek ponovimo še z oglišči okvira predmeta (angl. *object bounds*), pri vsakem od njih pa pred tem še preverimo kot (2. korak).

   V tem koraku pride do nenatančnosti zaradi aproksimacije oblike predmeta. Okvir predmeta je v svetu predstavljen kot kvader, ravno dovolj velik, da je v njem vsebovan celoten predmet. Sicer bi bilo število oglišč (in posledično število metov žarka) preveliko. Poleg tega so trkalniki (angl. *colliders*) objektov v svetu večinoma približki tistega, kar dejansko vidimo (priljubljene so krogle, kvadri, kapsule in valji), saj bi bila sicer zaznava trkov (angl. *collision detection*) prezahtevna.

   Če katerikoli žarek doseže predmet, rečemo, da je le-ta viden (čeprav je v resnici lahko viden le zelo majhen del predmeta).

Povzetek implementacije vida najlažje prikažemo v psevdokodi (slika 5.1).

5.2 Prvi scenarij

V prvem scenariju bomo implementirali tri različne agente:

- Preprost odzivni agent
- Agent s stanji
- Agent s spominom

V razdelku 4.2 smo opisali lastnosti scenarija in ugotovili, da je problematična le delna vidnost scenarija. Zato bo že preprost odzivni agent razmeroma dober približek inteligence (glede na preprostost), nadgradnji pa mu bosta obnašanje še močno popravili. Zadnji – agent s spominom – bo skoraj povsem

Slika 5.1: Implementacija vida.

Input: camera, item, max_distance
Output: item_visible: *boolean*

```
if |camera.position − item.position| > max_distance then
    return false
for vertex ∈ ({item.position} ∪ item.bounding_box.vertices) do
    forward_vector = camera.looking_direction_vector
    item_vector = vertex − camera.position
    if (
        HorizontalAngle(forward_vector, item_vector) >
            camera.FOV.horizontal_angle / 2 and
        VerticalAngle(forward_vector, item_vector) >
            camera.FOV.vertical_angle / 2 and
        RaycastSuccessful(camera.position, vertex)
    ) then
        return true
return false
```

odpravil problem delne vidnosti, tako da se bo v tem scenariju obnašal že bolj ali manj popolnoma inteligentno – vedno bo izbral akcijo, ki mu bo maksimizirala možnost uspeha glede na predznanje in zaporedje zaznav.

5.2.1 Preprost odzivni agent

Začnemo s preprostim odzivnim agentom. Ta bo kamero premikal naključno z določeno hitrostjo, če pa bo tarča v vidnem polju, bo hitrost zmanjšal in začel streljati. Hitrost spreminja vsake 3 sekunde (če bi jo spreminjal preveč pogosto, npr. na vsakem koraku, bi se zataknil že takoj na začetku). Omejeno ima vertikalno komponento pogleda, tako da se vertikalna komponenta smeri obrne, če pogleda preveč gor ali dol.

Agent torej hrani nekaj podatkov o svojem notranjem stanju (čas zadnje menjave smeri in trenutno smer), o stanju sveta pa ve le to, kar zaznava v tistem trenutku.

Tak agent je seveda zelo preprost (glej sliko 5.2), vseeno pa bo tarče občasno zadel (kar je glede na preprostost implementacije že precej dobro obnašanje). Seveda bi lahko uporabili še množico dodatnih če-potem pravil, da bi mu obnašanje izboljšali (kot rečeno, se da tudi agenta s stanji realizirati v obliki če-potem pravil), vendar pa s tem izgubimo pravzaprav edino lepo lastnost, ki jo ta agent ima: preprostost.

5.2.2 Agent s stanji

Sedaj agenta nadgradimo v avtomat s stanji, kar nam omogoča že mnogo bolj inteligentno obnašanje. Agent se bo zdaj že povsem inteligentno odzival na zaznave, popolno uporabo predznanja pa bomo dodali šele v naslednjem razdelku. Agent ima tri stanja med katerimi bo prehajal, vidimo jih na sliki 5.3.

Iskanje tarče: Agent deluje podobno kot SRA, ko ni videl tarče, le da je sedaj naključna le vertikalna komponenta premikanja. Agent se namreč

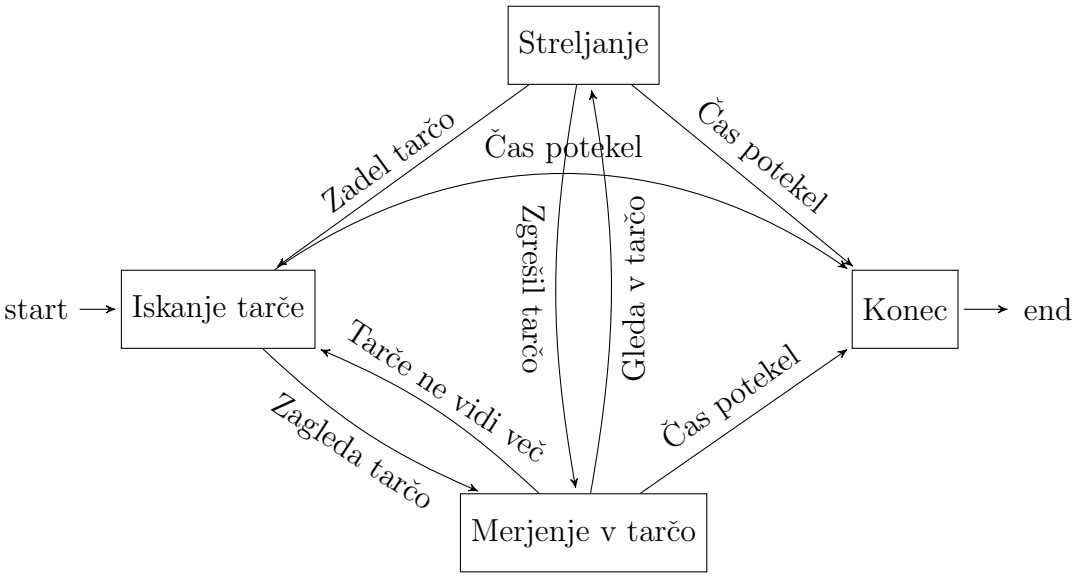
Slika 5.2: Preprost odzivni agent – potek razmišljanja na vsakem koraku.

Internal: direction, last_change_time

Percept: *sight*, current_time

---

if current_time > last_change_time + 3 **then**
    direction = GetRandomDirection()
    last_change_time = current_time
if TargetVisible() **then**
    Shoot()
    UseLowSpeed()
else
    UseHighSpeed()
MoveCamera(direction)



Slika 5.3: Agent s stanji.

sedaj lahko (ker je stanje namenjeno izključno iskanju tarče) vrti stalno v isto smer (desno).

Ko zagleda tarčo, preide v stanje merjenja v tarčo.

Merjenje v tarčo: Agent premika kamero proti tarči. Za lepši izgled pri premikanju hitrost prilagajamo oddaljenosti od tarče (s tem je obnašanje tudi bolj podobno človeškemu vodenju, saj prav tako v prvoosebnih igrah merimo ljudje – dokler smo daleč od cilja, premikamo miško sunkovito, ko pa pridemo blizu, pa se poslužimo počasnejših, natančnejših premikov). Prilagajanje hitrosti rotacije izvedemo s sferično interpolacijo med trenutno in ciljno rotacijo kamere.

Ko agent gleda naravnost v tarčo, preide v stanje streljanja, če pa vmes tarče ne vidi več (ker je preteklo 30 sekund in je tarča izginila), pa preide nazaj v stanje iskanja tarče.

Streljanje: Agent ustreli in če zadane tarčo, preide v stanje iskanja, če pa jo zgreši, pa v stanje merjenja. Agent lahko tarčo zgreši, saj streljanje ni povsem natančno – v osnovi strelja v smeri pogleda, vendar pa strel lahko odstopa od le-te za nekaj stopinj gor/dol in levo/desno. V primeru zgrešitve bo seveda kmalu spet prišel v stanje streljanja in poskusil znova.

Agent ima ponovno shranjenih nekaj podatkov o notranjem stanju, še vedno pa ne ve nič o zunanjem stanju razen tega, kar trenutno zaznava. Povzemimo obnašanje spet s psevdokodo na sliki 5.4.

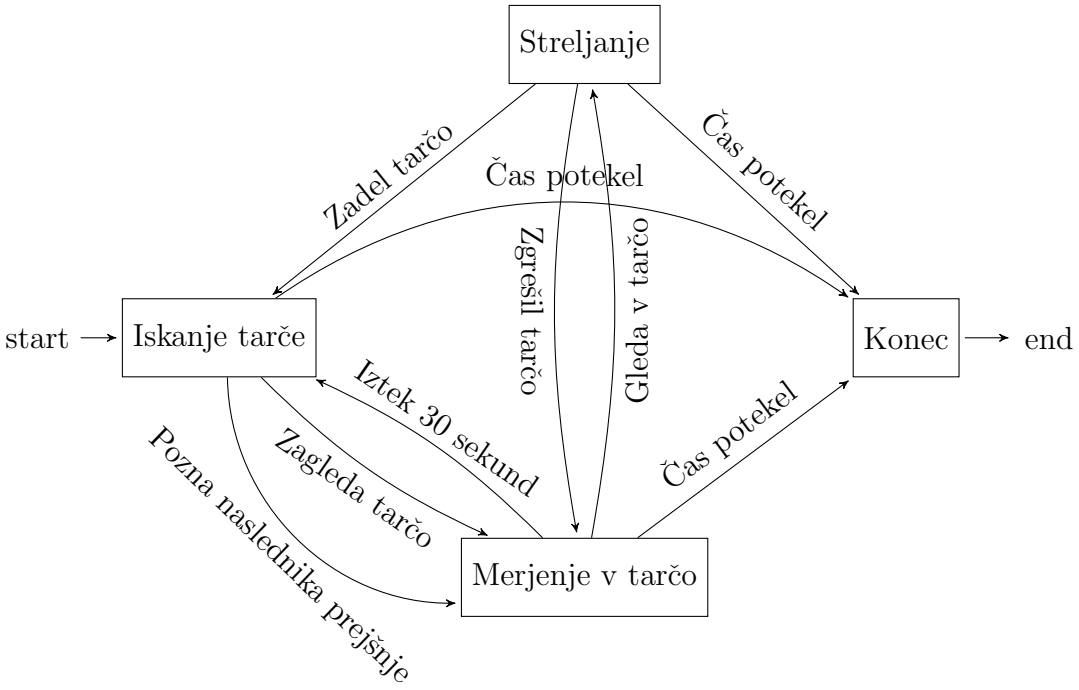
5.2.3 Agent s spominom

Na koncu agentu dodamo še spomin o stanju sveta. V opisu predznanja smo omenili, da agent ve, da se bodo tarče vedno pojavljale v enakem vrstnem redu, ne ve pa, koliko jih je in kakšen je ta vrstni red. Zato si vsakič, ko zagledamo novo tarčo, le-to zapomnimo kot naslednika prejšnje (če o prejšnji

Slika 5.4: Korak razmišljanja agenta s stanji.

Internal : state, vertical_direction, last_change_time
Percept : *sight*, *hearing*
Initialize: state = Searching

```
switch state do
case Searching
    MoveCameraRight()
    MoveCameraUpDown(vertical_direction)
    ChangeDirectionIfNeeded(vertical_direction, last_change_time)
    if TargetVisible() then
        state = Aiming
case Aiming
    MoveCameraTowardsTarget()
    if LookingDirectlyAtTarget() then
        state = Firing
    else if not TargetVisible() then
        state = Searching
case Firing
    Shoot()
    if TargetHit() then
        state = Searching
    else
        state = Aiming
```



Slika 5.5: Agent s spominom.

imamo informacijo – ta se izbriše po 30 sekundah, saj sta se v tem primeru vmes tarči že zamenjali, prav tako pa ni na voljo čisto na začetku).

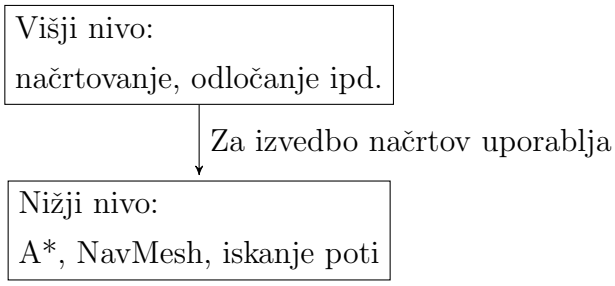
Stanja ohranimo enaka, popravimo le prehode, kot vidimo na sliki 5.5. Sedaj gremo iz stanja iskanja tarče v stanje merjenja, tudi če tarče ne vidimo, če le imamo informacijo o nasledniku prejšnje, iz stanja merjenja pa v tem istem primeru ne gremo nazaj v stanje iskanja, pač pa ta prehod naredimo le v primeru, ko se izteče 30 sekund. Dopolnjena psevdokoda je vidna na sliki 5.6.

Tak agent se bo obnašal inteligentno, saj bo uporabil vse svoje znanje (tako predznanje kot tudi zgodovino zaznav) zato, da bo maksimiziral svojo mero uspeha.

Slika 5.6: Korak razmišljanja agenta s spominom.

Internal : state, vertical_direction, last_change_time,
next_target_change_time
Memory : successors, previous, current
Percept : current_time, *sight*, *hearing*
Initialize: state = Searching

```
switch state do
case Searching
    HandleCameraMovement(vertical_direction, last_change_time)
    if previous.exists and successors.HasSuccessorOf(previous) then
        current = successors.SuccessorOf(previous)
        state = Aiming
    else if TargetVisible() then
        current = visible target
        if previous.exists then
            successors.Add(current as successor to previous)
        state = Aiming
case Aiming
    MoveCameraTowardsTarget()
    if current_time > next_target_change_time then
        previous = current
        next_target_change_time += 30
        state = Searching
    else if LookingDirectlyAtTarget() then
        state = Firing
case Firing
    Shoot()
    if TargetHit() then
        previous = current
        next_target_change_time = current_time + 30
        state = Searching
    else
        state = Aiming
```



Slika 5.7: Arhitektura agentov za drugi scenarij.

5.3 Drugi scenarij

V drugem scenariju bodo agenti imeli dva sloja: prvi, višjenivojski, bo skrbel za odločanje in načrtovanje, drugi, nižjenivojski, pa bo skrbel le za iskanje poti med točkami z algoritmom A*. Skico naše arhitekture najdemo na sliki 5.7. Nižji nivo bo realiziran predvsem z uporabo metod, ki so že na voljo v Unityjevem pogonu, višji nivo pa bomo v celoti implementirali sami in bo predstavljal glavno razliko med delovanjem posameznih agentov. Izjema je preprost odzivni agent, ki bo seveda implementiran kar najbolj enostavno.

Implementirali bomo naslednje agente:

- Preprost odzivni agent
- Preprost ciljno usmerjen agent
- Ciljno usmerjen agent s stanji
- Ciljno usmerjen agent s spominom
- Ciljno usmerjen agent z avtonomnim raziskovanjem
- Agent s funkcijo koristnosti

Tokrat je neugodnih lastnosti več – delna vidnost, stohastičnost, sekvenčnost. Kot smo že omenili, bomo zaradi sekvenčnosti na nivoju iskanja poti med točkami (nižji nivo) takoj potrebovali ciljno usmerjenost. Vseeno najprej implementiramo preprostega odzivnega agenta, prav zato da pokažemo,

zakaj je ciljna usmerjenost potrebna. Nato ciljno usmerjenost dodamo in potem začnemo nadgrajevati višji nivo. Najprej mu dodamo stanja, nato spomin, nato avtonomno raziskovanje in zaključimo z implementacijo agenta s funkcijo koristnosti.

Preden začnemo z implementacijo agentov, pa najprej povejmo nekaj o kamerah, ki so nam na voljo v drugem scenariju.

5.3.1 Postavljanje kamer v svetu

Prvoosebna kamera: Ta se nahaja na položaju oči našega lika in se z njim premika in vrti. Za tako obnašanje poskrbi že Unity, če kamero v sceno dodamo kot sestavni del našega lika, zato s postavljanjem te kamere ni težav. To kamero bomo uporabili kot kamero pogleda pri testiranju vidnosti.

Kamera OTS: Tretjeosebna kamera čez ramo je ob zagonu scenarija izbrana samodejno. Locirana je za likom, nekoliko dvignjena, usmerjena proti ramenom igralca. Ob vrtenju lika se rotaciji prilagodi postopoma in ne skače (tu spet uporabimo sferično interpolacijo med rotacijo kamere in rotacijo lika, stalno pa ji medtem popravljamo položaj). Položaj kamere določimo s sfernimi koordinatami, glede na igralca. Te nam omogočajo, da kamero vedno postavimo pravilno, ne glede na njen naklon in to, kam je obrnjena (tudi kadar ni obrnjena v isto smer kot igralec, jo lahko brez težav postavimo tako, da bo usmerjena proti njemu). To nam omogočata sferna kota (azimut in elevacija oz. θ in φ), radij pa uporabimo za približevanje in oddaljevanje kamere od lika.

Izometrična kamera: Gre za ortografsko kamero, kar pomeni, da njena slika ni perspektivno popačena in bodo predmeti enako veliki, ne glede na to, kako daleč so od kamere. Zato pri postavitvi ne skrbimo za njeno oddaljenost in jo preprosto postavimo daleč od igralca. Efekt približevanja in oddaljevanja dosežemo s povečanjem in pomanjšanjem leče kamere, s čimer spremenimo, koliko bo kamera zajela v svoj pogled.

Pri postavitvi pa moramo še vedno paziti, da jo oddaljimo od igralca v pravi smeri, zato uporabimo enak pristop s sfernimi koordinatami kot pri tretjeosebni kameri.

5.3.2 Preprost odzivni agent

Preprostega odzivnega agenta bomo v tem scenariju uporabili le za prikaz, kako slabo se obnese v takem okolju. Deloval ne bo niti približno inteligentno, saj bo akcije izbiral z vidika naloge in mere uspeha povsem naključno in pri izbiri ne bo smiselno uporabljal niti predznanja niti zaznav.

Preprost odzivni agent se po svetu premika naravnost, ko pa se zaleti v oviro na tak način, da se mu hitrost preveč zmanjša (približno čelno trčenje), pa smer naključno spremeni (obrne se za $90 - 270°$). Ker ima na voljo le senzor za položaj, ne pa tudi za hitrost, si od notranjega stanja zapomni položaj v prejšnjem koraku in nato za oceno hitrosti uporabi oddaljenost od njega in pretekel čas od prejšnjega koraka.

Povzetek delovanja je v psevdokodi na sliki 5.8.

Slika 5.8: Korak preprostega odzivnega agenta.

Internal: last_position
Percept: current_position, time_from_last_update

MoveForward()
if |current_position − last_position| < time_from_last_update **then**
    Rotate(Random(90,270))
last_position = current_position

Tega agenta bi se sicer dalo izboljšati podobno kot v prvem scenariju z dodatnimi pravili, kar pa, kot že rečeno, ni smisel preprostega odzivnega agenta. Prav tako bi lahko izboljšali obnašanje s stanji in spominom (to bomo tudi storili nekoliko kasneje), vendar pa raje najprej dodamo veliko pomembnejšo in bolj učinkovito izboljšavo – ciljno usmerjenost.

5.3.3 Preprost ciljno usmerjen agent

Na višjem nivoju bo agent sicer še vedno deloval kot preprost odzivni agent (torej po vnaprej definiranih pravilih), vendar pa mu bomo dodali nov sloj s ciljno usmerjenostjo, ki mu bo omogočala premikanje med točkami v svetu. Na kakšen način bo iskanje poti realizirano, bomo opisali v nadaljevanju, najprej pa si poglejmo, kako bo deloval višji nivo.

Zaenkrat predpostavimo, da znamo med poljubnima točkama poiskati pot (če le-ta obstaja). Najpreprostejši način izpeljave scenarija je, da agentu kar sami določimo cilje, ki ga bodo popeljali po poti skozi okolice stojal. Na ta način bo scenarij opravil, ob sprehodu skozi okolice pa bo verjetno pobral tudi vsaj nekaj sekir. Tega pa ne moremo zagotoviti, razen če mu podamo cilje res zelo na gosto. To pa lahko povzroči, da se scenarij konča še preden končamo sprehod. Skica delovanja je v psevdokodi na sliki 5.9. Točke ob stojalih agent pridobi ob zagonu scenarija in jih doda na prave položaje.

Slika 5.9: Korak osnovnega ciljno usmerjenega agenta.

Knowledge: waypoints
Internal : current_waypoint
Initialize : waypoints.Add(FindRackLocations() to correct indices),
current_waypoint = waypoints.first

```
MoveToward(current_waypoint)
if GoalReached(current_waypoint) then
    if waypoints.next.exists then
        current_waypoint = waypoints.next
    else
        EndScenario()
```

Tak pristop je mogoč le, če kot programerji že vnaprej vemo, kje bodo stojala postavljena in kje bodo njihove okolice. Definicija scenarija pa je zagotovila le, da bo ta informacija na voljo agentu ob zagonu scenarija, ne

pa nujno nam ob implementaciji agenta. Zato si bomo pogledali tudi, kako se ta isti agent z enakimi cilji obnese v premaknjenem scenariju, kjer so stojala in okolice premaknjeni na druge položaje. Pri tem moramo okolice seveda nekoliko popraviti, da so še vedno v dosegljivem delu sveta. Podobno primerjavo bomo naredili tudi pri naslednjih agentih.

Trenutni agent torej še vedno ne uporablja zaznav, uporablja pa vsaj programerjevo predznanje. Vendar pa, kot rečeno, če se parametri naloge spremenijo, to predznanje ne bo več ustrezno in agent se bo obnesel veliko slabše.

Sedaj pa si poglejmo glavno nadgradnjo tega razdelka, to pa je ciljna usmerjenost za iskanje poti. V ta namen bomo uporabljali algoritem A*, ki ga bomo pognali na navigacijski mreži (angl. *Navigation Mesh, Navigational Mesh – NavMesh*).

Algoritem A*

Algoritem A* je danes najbolj rabljen algoritem za iskanje poti, poleg tega pa tudi eden najbolj uporabljenih in preprostih hevrističnih algoritmov za splošne probleme preiskovanja prostora stanj [36]. Razlog za to so mnoge lepe lastnosti, ki jih ima, predvsem pa hitro delovanje in (ob določenih predpostavkah) zagotovitev najkrajše poti.

Algoritem deluje na grafu $G = (\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ je množica vozlišč (točk) grafa, $\mathcal{E} = \{(u, v);\ u, v \in \mathcal{V}\}$ pa je množica povezav med vozlišči. Pari točk (u, v), ki te povezave predstavljajo, so lahko urejeni (tedaj je graf usmerjen) ali pa neurejeni (tedaj je graf neusmerjen).

A* je nadgradnja Dijkstrovega algoritma za iskanje najkrajših poti v grafu. Temu algoritmu doda hevristično funkcijo, deluje pa podobno kot t.i. iskanje najboljši najprej (angl. *best-first search*).

Hevristična funkcija vsakemu vozlišču grafa priredi neko oceno oddaljenosti od ciljnega vozlišča (boljša ko je hevristična funkcija, bolje bo algoritem deloval). To potem prištejemo ceni poti, ki smo jo že opravili od začetnega vozlišča do trenutnega, in tako dobimo ceno vozlišča. Razvijemo vedno voz-

lišče z najmanjšo ceno in nato njegovim potomcem izračunamo hevristično oceno in ceno. Postopek nadaljujemo dokler ne najdemo ciljnega vozlišča. Če za hevristično funkcijo vzamemo $\forall v \in \mathcal{V} : h(v) = 0$, dobimo ravno Dijsktrov algoritem.

Če je hevristična funkcija **sprejemljiva** (angl. *admissible*), torej v nobenem vozlišču ne preceni oddaljenosti od cilja, ta algoritem zagotavlja najkrajšo pot.

Hevristična funkcija je **monotona**, če velja $\forall (u, v) \in \mathcal{E} : h(u) \leq c(u, v) + h(v)$, kjer je $c(u, v)$ cena povezave med vozliščema (Russel in Norvig [32] definicijo zapišeta nekoliko drugače z nasledniki vozlišč, vendar pa koncept ostaja enak). Z drugimi besedami, monotona je, če razlika med hevrističnima ocenama poljubnih sosednih vozlišč nikoli ne preceni dejanske cene povezave med njima. Če je hevristična funkcija monotona, je implementacija A* učinkovitejša, saj nam zagotavlja, da vsako vozlišče lahko obiščemo največ enkrat.

Za evklidsko razdaljo, ki se v problemu iskanja poti v 3D svetu kot hevristična funkcija uporablja največkrat (uporabljena je tudi v Unityju), veljata obe zgornji lastnosti, zato je že Unityjeva implementacija algoritma A* zelo učinkovita.

Kako pa naj sedaj ta algoritem uporabimo v zveznem prostoru, ki ga predstavljajo 3D svetovi? Jasno je, da ne moremo uporabiti vsake možne pozicije v svetu kot vozlišča v grafu stanj, saj bi bil le-ta tako praktično neskončen. Potrebujemo nek smiseln način izbire točk, ki jih bomo uporabili za vozlišča in tu pride v pomoč Unityjeva struktura NavMesh.

Navigacijska mreža

NavMesh [11] je struktura, ki se v zadnjih 15 letih veliko uporablja za pomoč pri navigaciji v vnaprej poznanih modelih 3D svetov – najbolj očitna uporaba je v 3D svetovih iger, ki so seveda večinoma vnaprej poznani.

Obstajajo sicer načini, kako to strukturo približno graditi sproti (angl. *on-the-fly building, online building*), večinoma pa se NavMesh zgradi pred

zagonom scenarija (angl. *offline*), saj je gradnja NavMesha računsko zahteven proces, poleg tega pa pri sprotni gradnji pogosto postane pretirano kompleksen. Gradnja pred zagonom pa seveda zahteva vnaprej znano postavitev sveta.

Postopkov, kako NavMesh zgradimo, je več, prav tako pa se sam NavMesh lahko nekoliko razlikuje od izvedbe do izvedbe [34].

Celoten svet najprej zreduciramo na prehodne površine (angl. *walkable surfaces*) – te so približek tal, po katerih se agent lahko premika. Pod ovirami, skozi katere agent ne more, prehodne površine ni. Prav tako prehodne površine ni tam, kjer je premajhna razdalja od tal do stropa (lahko imamo tudi različne podatke za različna stanja igralca – v počepu, stoječ, v skoku ipd.).

Pred tem ponavadi ovire napihnemo za polovico širine našega lika. Tako dobimo na koncu površino, po kateri lahko namesto agenta premikamo točko, ki se nahaja na sredini njegovih podplatov. Točkast agent nam namreč olajša kasnejše postopke iskanja poti.

Tu predpostavimo, da je lik enako širok tako v širino kot v globino, torej da ni pomembno, v katero stran je obrnjen. To seveda včasih ne drži in možna bi bila uporaba pristopov iz robotike, kjer tako strukturo ponavadi razširijo v tretjo dimenzijo, ki predstavlja rotacijo agenta okoli navpične osi. Mi smo sicer že uvedli tretjo dimenzijo (predstavlja zgoraj omenjeno stanje igralca), vseeno pa bi lahko uporabili omenjeni pristop. Vendar pa ta močno poveča kompleksnost celotne strukture in algoritmov, ki jih uporabljamo, zato se ga v NavMeshih ne uporablja.

Prehodne površine nato predstavimo z večkotniki. Najprimernejši so, kot pogosto v predstavitvah 3D sveta v računalništvu, trikotniki. Ko enkrat dobimo tak razcep na trikotnike (triangulacijo), lahko definiramo graf, ki ga bomo uporabili v algoritmu A*. Za vozlišča grafa vzamemo kar oglišča trikotnikov, za povezave pa njihove robove. Ker so ti robovi vedno v celoti del NavMesha in posledično prehodnih površin, bo med njimi vedno možna

pot naravnost.[2] Zato je med stanji prehod trivialen (ne potrebuje dodatnega iskanja poti), za ceno teh povezav pa lahko vzamemo kar dolžine robov. Na takem grafu sedaj lahko uporabimo algoritem A*, ki nam zagotavlja najkrajšo pot (če ta obstaja) v razumnem času.

Gradnjo NavMesha (angl. *NavMesh baking*) nam omogoča že Unity, vendar za ovire vzame izrisovalce (angl. *renderers*), ne pa trkalnikov (angl. *colliders*). Torej upošteva vidne predmete, ne pa njihovih trkalnikov, ki se v igri uporabljajo za detekcijo trkov našega lika s svetom. Zato je potrebnega nekaj dodatnega dela s strani razvijalca, preden prepusti gradnjo Unityju.

5.3.4 Ciljno usmerjen agent s stanji

Tako smo implementirali nižji nivo agenta in se lahko posvetimo nadgradnjam na višjem. Podobno kot v prvem scenariju najprej dodamo stanja (to je sploh v igrah vedno dober prvi korak). Skico teh vidimo na sliki 5.10.

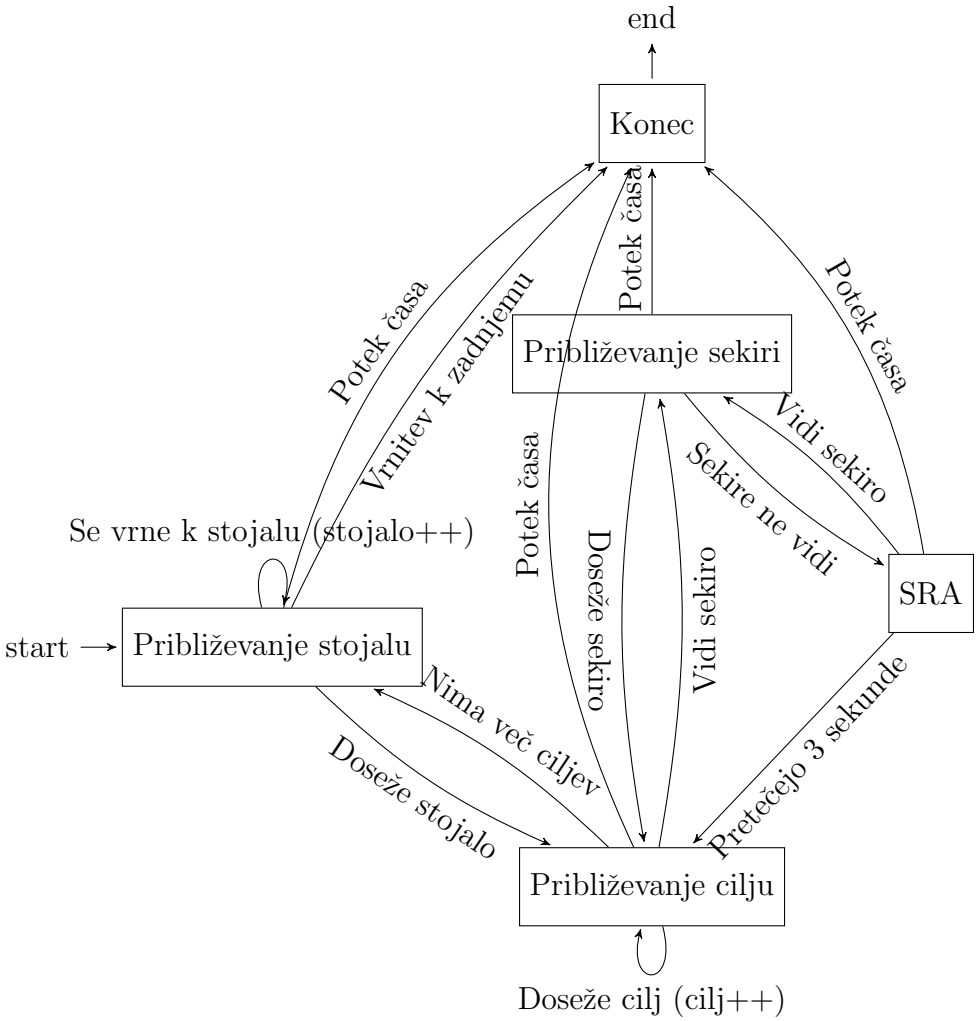
Približevanje stojalu: Agent se premika proti stojalu – pot stalno planira s pomočjo A*. Ob tem položaje stojal pridobi ob začetku scenarija, kot smo to omenili v prejšnjem razdelku.

Ko doseže stojalo, preide v približevanje cilju. Pred tem prebere, katere ciljne točke mu je programer definiral v okolici tega stojala. Ko pa se do stojala vrne (po obhodu ciljnih točk), pa se začne približevati naslednjemu. Če naslednjega stojala ni, agent scenarij zaključi.

Približevanje cilju: Agent se premika proti trenutni ciljni točki. Ponovno pot načrtuje s pomočjo A*.

Če doseže ciljno točko, nadaljuje proti naslednji, če pa je bila dosežena zadnja v trenutni okolici, pa preide v stanje približevanja stojalu (se vrača k stojalu). Če kadarkoli vmes vidi sekiro, preide v približevanje sekiri.

[2] V resnici za povezave v grafu vzamemo več kot le robove trikotnikov, saj ti omogočajo le prehod med sosednjimi točkami (oglišči istega trikotnika). Za povezave vzamemo vse pare točk, pri katerih je pot naravnost med točkama popolnoma vsebovana v NavMeshu.



Slika 5.10: Ciljno usmerjen agent s stanji.

SRA: V stanju SRA se agent obnaša tako kot preprost odzivni agent, s katerim smo začeli (torej hodi naravnost, se odbija od sten). Potrebo po tem stanju bomo videli kmalu.

To stanje traja 3 sekunde, po katerih preide nazaj v stanje približevanja cilju. Če vmes zagleda sekiro, preide v približevanje sekiri.

Približevanje sekiri: Agent se premika proti vidni sekiri in si pri tem spet pomaga z A*.

Če doseže cilj (torej sekiro pobere), preide nazaj v približevanje cilju in nadaljuje svoj obhod. Če pa sekiro izgubi iz pogleda, pa preide v stanje SRA. To je potrebno, ker agent kamere pogleda ne premika gor in dol, zato bo, ko pride zelo blizu sekire, to pogosto izgubil iz vidnega polja. Ker v scenariju tako vid kot tudi preverjanje dosega ciljne točke delujeta približno, se bo včasih to zgodilo, včasih pa bo agent že pred tem dosegel ciljno točko. Če sekiro izgubi iz vidnega polja, potem v stanju SRA lahko nadaljuje naravnost še nekaj časa, kar mu bo v primeru, ko je dovolj blizu, omogočilo, da sekiro pobere. Prav tako mu to stanje lahko pomaga, če ga algoritem A* obrne preveč stran od sekire ali pa ga popelje mimo kakšne ovire, ki mu zastre pogled na sekiro. Ko sekiro v stanju SRA zagleda, preide nazaj v približevanje sekiri (ta ni nujno ista, kot smo se ji približevali prej, saj lahko vmes zagleda kakšno drugo).

Povzemimo obnašanje spet s psevdokodo na sliki 5.11.

Agent sedaj uporablja tudi zaznave in se tako ne zanaša več samo na predznanje (ki je bilo predvsem programerjevo, ne agentovo). Zato bo nalogo opravil bolj zanesljivo, nekoliko bolje pa se bo znašel tudi v premaknjenem scenariju.

5.3.5 Ciljno usmerjen agent s spominom

Nadaljujemo podobno kot v prvem scenariju in agentu dodamo še spomin o stanju sveta. Ko agent opazi sekiro, si njen položaj zapomni in ga uporabi kot

Slika 5.11: Korak ciljno usmerjenega agenta s stanji.

Knowledge: given_waypoints

Internal : state, waypoints, current_waypoint, racks, current_rack

Percept : *sight*

Initialize : racks = FindRackLocations(), current_rack = racks.first

---

```
switch state do
case ApproachingRack
    MoveToward(current_rack)
    if GoalReached(current_rack) for the first time then
        waypoints = GetWaypoints(current_rack, given_waypoints)
        current_waypoint = waypoints.first
        state = ApproachingWaypoint
    else if GoalReached(current_rack) again then
        if racks.next.exists then
            current_rack = racks.next
        else
            EndScenario()
case ApproachingWaypoint
    MoveToward(current_waypoint)
    if AxeVisible() then
        state = ApproachingTool
    else if GoalReached(current_waypoint) then
        if waypoints.next.exists then
            current_waypoint = waypoints.next
        else
            state = ApproachingRack
case ApproachingTool
    MoveToward(visible axe)
    if not AxeVisible() then
        state = SRA
    else if GoalReached(visible axe) then
        state = ApproachingWaypoint
case SRA
    MoveForwardAndTurnIfBlocked()
    if AxeVisible() then
        state = ApproachingTool
    else if 3 seconds have passed then
        state = ApproachingWaypoint
```

dodaten cilj z višjo prioriteto. Tako ne bo imel več potrebe po stanju SRA in ne bo imel težav, če bo imel v vidnem polju več kot eno sekiro. Poleg tega si zapomni, koliko sekir je do sedaj pobral in, če pobere vseh 5, prekine obhod okolice in nemudoma preide v približevanje stojalu. V psevdokodi (slika 5.12) prikažemo le stanji približevanja ciljem in sekiram, saj le pri njiju pride do sprememb (v približevanju stojalu je potrebno le še dodati resetiranje števila pobranih sekir ob začetku obhoda na 0).

5.3.6 Nadgradnja z avtonomnim raziskovanjem

Dosedanji agenti so sicer imeli ciljno usmerjenost na nivoju sprehajanja med točkami, na nivoju celotnega scenarija pa se je agent še vedno obnašal povsem po vnaprej določenih pravilih, zato odpove, če nekoliko spremenimo parametre naloge (premaknemo stojala in okolice). Zato agentu dodamo ciljno usmerjenost še na višjem nivoju in s tem agentu dodamo avtonomnost, ki je zelo zaželena lastnost agentov. Agent deluje podobno kot prejšnji, razlikuje pa se v fazi preiskovanja okolice stojala. Prej je to počel z obhodom vnaprej določenih ciljnih točk, ki mu jih je za tisto stojalo določil programer, sedaj pa si bo te točke znal postaviti sam. Cilj, ki ga s ciljno usmerjenostjo poskuša doseči, je popolna pokritost celotne okolice – to pomeni, da med preiskovanjem vsaj enkrat vidi vsako točko v okolici.

V definiciji scenarija smo povedali, da so okolice, v katerih se lahko pojavijo sekire, definirane s kvadri, ki so agentu poznani ob zagonu scenarija. Nikjer niso presekani s kakšno oviro, tako da s katerekoli točke v določenem kvadru lahko vidimo vse preostale točke tega kvadra. Zato je za preiskovanje dovolj, če iz vsakega kvadra iz okolice vzamemo naključno izbrano točko, se premaknemo do nje in se na njej obrnemo okoli. Sekire v tem kvadru tako ne bomo opazili le, če je zelo blizu te točke (saj kamere ne obračamo gor in dol), v tem primeru pa ga bomo tako ali tako ob prihodu na točko pobrali ali pa opazili že prej.

Ker vid in navigacija nista povsem natančna, moramo v sami aplikaciji upoštevati možnost, da ne bomo našli vseh sekir v prvem obhodu, zato ob

Slika 5.12: Korak ciljno usmerjenega agenta s spominom (le spremembe glede na sliko 5.11).

Knowledge: given_waypoints
Internal : state, waypoints, current_waypoint, racks, current_rack
Memory : priority_waypoints, tools_picked_up
Percept : *sight*
Initialize : racks = FindRackLocations(), current_rack = racks.first

---

```
switch state do
case ApproachingWaypoint
    MoveToward(current_waypoint)
    if GoalReached(current_waypoint) then
        if waypoints.next.exists then
            current_waypoint = waypoints.next
        else
            state = ApproachingRack
    if AxeVisible() then
        priority_waypoints.Add(visible axe)
        state = ApproachingTool
case ApproachingTool
    if AxeVisible() then
        priority_waypoints.Add(visible axe)
    MoveToward(priority_waypoints.first)
    if GoalReached(priority_waypoints.first) then
        tools_picked_up += 1
        priority_waypoints.RemoveFirst()
    if tools_picked_up == 5 then
        state = ApproachingRack
    else if priority_waypoints == ∅ then
        state = ApproachingWaypoint
```

vrnitvi do stojala pogledamo, če smo vrnili vseh 5 sekir in preiskavo okolice ponovimo, če nam to ni uspelo. Za preprostejši opis delovanja pa tukaj na to lahko pozabimo in predpostavimo, da postopek vedno že v prvem obhodu najde vse sekire.

Stanje približevanja sekiri ostane enako, zato ga v psevdokodi (slika 5.13) izpustimo. Sprememba v ostalih dveh stanjih je ta, da si ciljne točke sedaj sproti določamo po postopku, ki smo ga opisali zgoraj.

Agent se bo dobro znal prilagajati na spremembe v parametrih naloge, torej smo mu dodali lastnost avtonomnosti. Ta nadgradnja je prva, ki res zagotavlja, da bo agent scenarij (tudi premaknjenega) dobro opravil z vidika vrnjenih sekir, čeprav mogoče ne najhitreje.

5.3.7 Agent s funkcijo koristnosti

Zadnja nadgradnja, ki jo bomo izvedli, je nadgradnja na agenta s funkcijo koristnosti. Najprej si poglejmo, katere cilje je imel prejšnji agent na višjem nivoju: priti do vseh treh stojal, pri vsakem preiskati okolico (s popolno pokritostjo), v okolici pobrati vse sekire. Te cilje je sicer z gotovostjo dosegel, ni pa delal nobenih razlik med ciljnimi vozlišči, ki posamezne cilje predstavljajo, glede na to, kako dobri bodo z vidika mere uspeha in je izbral preprosto prvega, ki ga je našel. Za možnost izbire med cilji potrebujemo agenta s funkcijo koristnosti.

Lotimo se vsakega cilja posebej in določimo, kaj želimo, da agent doseže pri posameznem cilju za maksimiziranje celotne mere uspeha. V vsaki fazi torej definiramo funkcijo koristnosti (ni namreč nujno, da je ta med delovanjem agenta stalno enaka). Te faze pa so tri:

Sprehod med stojali: Nima vpliva na število točk, ki jih bomo zbrali, pač pa le na čas, ki ga bomo porabili. Tega torej želimo za čim boljšo mero uspeha čim bolj zmanjšati. Ker ima agent konstantno hitrost, bomo čas minimizirali, če minimiziramo opravljeno pot.

Ker se ob vsakem obisku stojala po obhodu okolice vrnemo nazaj do

Slika 5.13: Korak ciljno usmerjenega agenta z avtonomnim raziskovanjem (le spremembe glede na sliko 5.12).

Internal : state, boxes, current_waypoint, racks, current_rack
Memory : priority_waypoints, tools_picked_up
Percept : *sight*
Initialize: racks = FindRackLocations(), current_rack = racks.first

---

```
switch state do
case ApproachingRack
    MoveToward(current_rack)
    if GoalReached(current_rack) for the first time then
        boxes = GetBoxes(current_rack)
        current_waypoint = GetRandomPointInBox(boxes.first)
        state = Exploring
    else if GoalReached(current_rack) again then
        if racks.next.exists then
            current_rack = racks.next
        else
            EndScenario()
case Exploring
    MoveToward(current_waypoint)
    if GoalReached(current_waypoint) then
        if not boxes.next.exists or tools_picked_up == 5 then
            state = ApproachingRack
        else
            current_waypoint = GetRandomPointInBox (boxes.next)
    if AxeVisible() then
        priority_waypoints.Add(visible axe)
        state = ApproachingTool
```

njega, lahko fazo sprehoda med stojali res obravnavamo izolirano in torej minimiziramo njen prispevek k skupnemu času tako, da poiščemo najkrajši sprehod od začetne točke do vseh treh stojal.

Za funkcijo koristnosti torej lahko vzamemo kar dolžino poti sprehoda med stojali, pomnoženo z -1, torej

$$f(sprehod) = -d(sprehod)$$

Za njeno maksimiziranje bomo stojala uredili tako, da bo sprehod najkrajši. Tako pridemo do (nekoliko prirejenega) problema potujočega trgovca (angl. *travelling salesman problem – TSP*) s štirimi točkami (našo začetno in tremi položaji stojal). Kako bomo ta problem reševali, bomo opisali malo kasneje.

Preiskovanje okolice: Vpliva tako na število točk (če bomo okolico dobro preiskali, bomo našli vse sekire) kot tudi na čas. Za čim boljšo mero uspeha torej želimo v tej fazi maksimizirati pričakovano število najdenih sekir in minimizirati porabljen čas, poudarek pa je na številu najdenih sekir (ki je v meri uspeha bolj pomembno). Izbrati pa moramo v tej fazi ciljne točke v okolici in pot, ki jo bomo med njimi ubrali.

Če hočemo najti čim več sekir, moramo v tej fazi maksimizirati pokritost okolice, za minimizacijo porabljenega časa pa moramo spet, podobno kot prej, najti najkrajši sprehod med izbranimi ciljnimi točkami. Ker nam je število sekir pomembnejše, je še bolje, če v tej fazi kar zahtevamo popolno pokritost okolice.

Funkcijo koristnosti torej definiramo spet kot dolžino sprehoda med izbranimi ciljnimi točkami, dodamo pa zahtevo, da le-te popolnoma pokrijejo okolico. Definiramo jo torej lahko kot:

$$f(točke, sprehod) = \begin{cases} -d(sprehod); & \text{točke pokrijejo celo okolico} \\ -\infty; & \text{sicer} \end{cases}$$

Če bomo to maksimizirali, bomo res dobili najboljšo kombinacijo ciljnih točk in sprehoda med njimi, ki bo zagotovila popolno pokritost in minimizirala porabljen čas.

Ker pa je možnih izbir ciljnih točk ogromno (lahko izberemo poljubno število točk iz poljubnih kvadrov okolice), ne bomo mogli pregledati vseh možnih izbir, zato bomo uporabili dovolj dober približek najboljše izbire (postopek iskanja tega približka bomo opisali malo kasneje). Pri minimizaciji dolžine sprehoda spet srečamo enak problem kot v prvi fazi.

Pobiranje sekir: Ta faza spet vpliva na obe komponenti mere uspeha, vendar pa bomo, podobno kot v prejšnji fazi, zahtevali, da vse najdene sekire tudi pobere. Ker tukaj položaji sekir niso spremenljivi (prej so izbrane ciljne točke bile), lahko ta del kar izpustimo iz funkcije koristnosti in uporabimo spet enako kot v prvi fazi:

$$f(sprehod) = -d(sprehod)$$

Pobiranje sekir bomo vključili kar v preiskovanje okolice, tako da gre tu pravzaprav za tisti sprehod, ki smo ga našli v prejšnji fazi. Položaje sekir bomo dodali kar v sprehod sam, in sicer na tak način, da bomo minimizirali dodaten čas, ki ga bomo za pobiranje sekir porabili. To dosežemo tako, da vsakič, ko zagledamo novo sekiro, njen položaj dodamo kot novo ciljno točko takoj za ciljno točko preiskovanja, ki ji je najbližja. Če pa že poznamo položaje vseh še nepobranih sekir, se nemudoma lotimo pobiranja sekir (pred tem spet izračunamo najkrajšo pot med njimi – spet problem TSP).

Kot vidimo, do kompleksnih odločanj pride med načrtovanjem akcij. Načrtovanje pa se izvede le enkrat pred množico akcij, ki jih mora agent izvesti za dosego načrtovanega cilja. Zato ta nadgradnja ne bo usodno povečala računske zahtevnosti izvajanja agenta.

Preden se lotimo problema potujočega trgovca, opišimo še, kako se v fazi načrtovanja preiskovanja okolice odločimo za najboljšo kombinacijo ciljnih

točk. Najpreprostejše merilo (ki v praksi deluje dobro) je število izbranih ciljnih točk. Radi bi, da je to čim manjše, da pa vseeno zagotavljajo popolno pokritost. Za izbiro ciljnih točk bomo uporabili sledeč postopek:

1. Seznam kvadrov inicializiramo na kvadre v okolici.
2. Iz seznama kvadrov izberemo naključen kvader in ga iz seznama izbrišemo.
3. Izberemo naključno točko iz kvadra.
4. Iz seznama izbrišemo vse kvadre, ki so popolnoma vidni z izbrane ciljne točke. Popolno vidnost kvadra spet preverjamo približno, in sicer tako, da ciljno točko najprej dvignemo do višine oči, nato pa preverimo vidnost (podobno kot v implementaciji vida) vsakega od 8 oglišč kvadra. Če je vseh 8 oglišč vidnih, rečemo, da je kvader popolnoma viden, sicer pa ni.
5. Če je v seznamu še kakšen kvader, se vrnemo na 2. korak, sicer postopek končamo.

Tak postopek nam zagotavlja popolno pokritost okolice. Za približek najboljše izbire cel postopek nekajkrat (v naši implementaciji 20-krat) ponovimo in uporabimo rešitev z najmanj ciljnimi točkami.

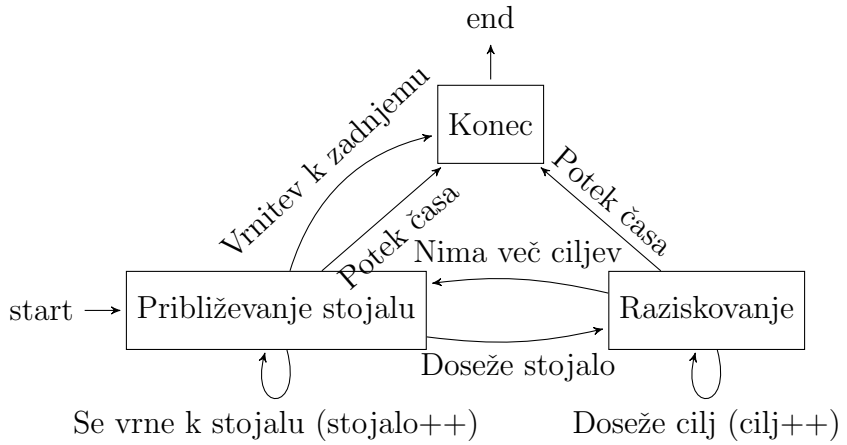
Poglejmo si sedaj delovanje agenta v psevdokodi na sliki 5.14 ob predpostavki, da znamo rešiti problem potujočega trgovca s funkcijo `SolveTSP`, ki nam ciljne točke uredi po vrstnem redu najkrajšega sprehoda. Podobno kot prej za preprostejšo predstavo zanemarimo možnost, da ob prvem obhodu okolice ne najdemo vseh sekir.

Opazimo, da v kodi funkcija koristnosti ni nikjer eksplicitno zapisana, jo pa na vsakem koraku (kot zgoraj opisano) poskušamo maksimizirati (kolikor nam to omogočajo računske sposobnosti računalnika) – uporabljamo jo implicitno. Poleg tega opazimo, da je tokrat spremenljivka `waypoints` del tako notranjega stanja (izbrane ciljne točke) kot tudi spomina (med ciljne točke

Slika 5.14: Korak agenta s funkcijo koristnosti.

Internal : state, waypoints, current_waypoint, racks, current_rack
Memory : waypoints, priority_waypoints, tools_picked_up
Percept : *sight*
Initialize: racks = FindRackLocations(), SolveTSP(racks),
current_rack = racks.first

```
switch state do
case ApproachingRack
    MoveToward(current_rack)
    if GoalReached(current_rack) for the first time then
        waypoints = GetBestExplorationWaypoints(current_rack)
        SolveTSP(waypoints)
        current_waypoint = waypoints.first
        state = Exploring
    else if GoalReached(current_rack) again then
        if racks.next.exists then
            current_rack = racks.next
        else
            EndScenario()
case Exploring
    MoveToward(current_waypoint)
    if GoalReached(current_waypoint) then
        TurnAround()
        if not waypoints.next.exists or tools_picked_up == 5 then
            state = ApproachingRack
        else
            current_waypoint = waypoints.next
    if AxeVisible() then
        priority_waypoints.Add(visible axe)
        closest = FindClosest(waypoints, visible axe)
        waypoints.Add(visible axe after closest)
        if priority_waypoints.count == 5 – tools_picked_up then
            waypoints = priority_waypoints
            SolveTSP(waypoints)
            current_waypoint = waypoints.first
```



Slika 5.15: Stanja agenta s funkcijo koristnosti.

dodamo položaje sekir, ko jih zagledamo). Ker pobiranje sekir realiziramo kar v fazi preiskovanja, sta za delovanje potrebni le še dve stanji, ki ju vidimo na sliki 5.15.

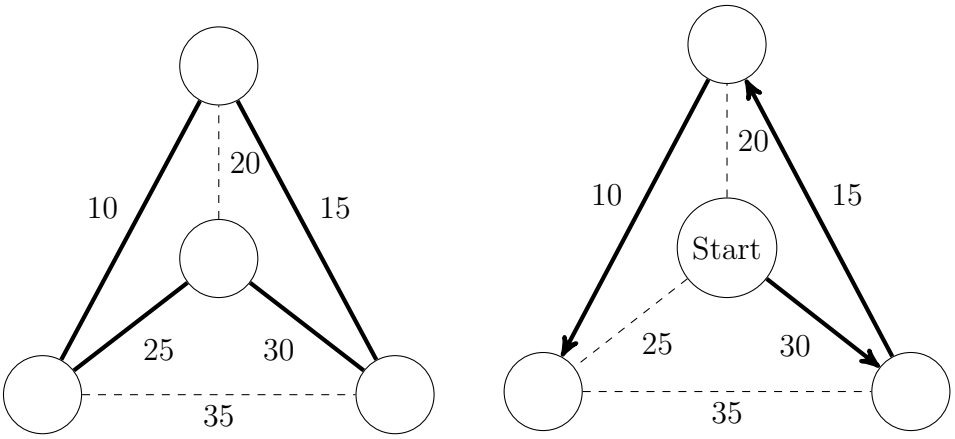
Agent se bo znal odločati med različnimi možnimi obhodi, še vedno pa bo ohranil popolno pokritost vseh okolic. Zato bo scenarij dobro opravil tako z vidika vrnjenih sekir kot tudi z vidika porabljenega časa.

Problem potujočega trgovca

V osnovni obliki problema potujočega trgovca moramo v grafu najti Hamiltonov cikel z najmanjšo težo. To pomeni, da moramo najti cikel (pot, ki se začne in konča v istem vozlišču), ki vsa vozlišča obišče natanko enkrat, in sicer tako, da bo vsota cen povezav, ki smo jih uporabili, najmanjša. Ta problem je NP-težak, kar pomeni, da eksakten polinomski algoritem zanj verjetno ne obstaja [15].

Naša variacija ne zahteva cikla, pač pa le Hamiltonovo pot, saj ni potrebe po vrnitvi na začetek.[3] Poleg tega imamo fiksirano začetno vozlišče na

[3]Po obhodu okolice se moramo sicer vrniti nazaj k stojalu, vendar moramo pred tem še pobrati sekire. Zato vračanja ne upoštevamo pri računanju najboljšega obhoda točk iz okolice.



Slika 5.16: a) Rešitev osnovnega TSP.
b) Rešitev naše variacije TSP na istem grafu.

agentovem trenutnem položaju. Slednje v osnovni obliki ne spremeni problema, saj lahko v ciklu poljubno spreminjamo začetno vozlišče, v primeru Hamiltonove poti pa je pomembno – obstajajo variacije s fiksnim začetnim in/ali končnim vozliščem. Prav tako obstajajo variacije za usmerjene grafe. Razliko naše variacije od osnovnega problema vidimo na sliki 5.16.

Za reševanje TSP bomo uporabili dva različna pristopa, odvisno od števila točk n. Za razdalje med točkami (cene povezav v grafu) uporabimo kar dolžine poti med njimi (poti izračunamo z A*), ne evklidskih razdalj.

$n \leq 7$: Pri dovolj majhnih vhodih (v naši implementaciji so to vhodi z manj kot 7 točkami) bomo preprosto pregledali vse možne permutacije točk (prva točka seveda ostane fiksna) in vrnili tisto z najmanjšo vsoto razdalj med njimi. Teh permutacij je $(n-1)!$, zato ta pristop ni primeren za večje število točk, je pa eksakten in preprost za implementacijo.

$n > 7$: Pri večjem številu točk bomo uporabili algoritem najbližjega soseda [20] (slika 5.17) in nato dobljeno pot popravili z optimizacijo 2-Opt [27] (slika 5.18).

Algoritem najbližjega soseda (angl. *nearest neighbor algorithm – NN algorithm*) je zelo preprosta rešitev. Gre za požrešni algoritem, ki začne z začetno točko (ta je pri nas fiksna) in nato vedno nadaljuje po povezavi z najmanjšo dolžino (ceno). Ta preprosta verzija algoritma bo seveda včasih odpovedala pri nepolnih grafih (ti k sreči v problemih iskanja poti v igrah niso pogosti). Prav tako pa ni eksakten. Obnese se razmeroma dobro v osnovni obliki problema, še boljše pa v naši variaciji problema, saj je prav povezava vračanja na začetno točko tista, ki mu v osnovnem problemu pogosto pokvari načrte.

Optimizacija 2-Opt je optimizacijska metoda za lokalno optimizacijo, razvita posebej za problem potujočega trgovca. Ideja, na kateri temelji, je, da se (ob predpostavki, da velja trikotniška neenakost – seveda to drži pri iskanju poti) da vsako sekanje poti odpraviti in tako dobiti boljšo rešitev. 2-Opt deluje tako, da poskuša obrniti vse možne pod-poti (spet brez začetnega vozlišča), tako, da se jih obišče v obratnem vrstnem redu. Če kakšna inverzija zmanjša dolžino celotne poti, vzame tako popravljeno celotno pot za novo in z njo spet ponovi postopek.

Postopek se ponavlja, dokler mu uspeva izboljšati pot, za vsak slučaj pa ga v implementaciji še omejimo na 30 korakov, da ne porabi preveč računskega časa.

S tema dvema algoritmoma rešimo primer s slike 5.16 (ki sicer nima več kot 7 točk, a kljub temu dobro prikaže delovanje algoritmov). NN nam vrne rešitev -20-10-35- (oglišč nimamo označenih, zato zapišemo s povezavami), ki ni optimalna.

Na njej sedaj izvedemo 2-Opt.

- Lahko zamenjamo 2. in 3. oglišče, obrnemo pot skozi 2., 3. in 4. ali pa zamenjamo 3. in 4 oglišče.

- Začnemo z zamenjavo 2. in 3. in dobimo pot -25-10-15-, ki ima manjšo skupno dolžino. To pot vzamemo kot novo in spet začnemo od začetka.
- Zamenjamo 2. in 3. oglišče in dobimo nazaj pot -20-10-35-, ki smo jo imeli prej. Ta je daljša in jo zavržemo – še vedno imamo pot -25-10-15-.
- Obrnemo pot skozi 2., 3. in 4. oglišče. Dobimo pot -30-15-10-, ki je krajša. Vzamemo jo kot novo in začnemo od začetka.
- Zamenjamo 2. in 3. oglišče in dobimo pot -20-15-35-. Ta je daljša, jo zavržemo. Še vedno imamo pot -30-15-10-.
- Obrnemo pot skozi 2., 3. in 4. oglišče in dobimo nazaj pot -30-15-10-, ki je daljša. Jo zavržemo, še vedno imamo pot -30-15-10-.
- Zamenjamo 3. in 4. oglišče in dobimo pot -30-35-10-. Ta je spet daljša in jo zavržemo.
- Nismo uspeli najti nobene izboljšave, zato postopek zaključimo s potjo -30-15-10-.

Kot vidimo na sliki, je rešitev, ki nam jo je postopek vrnil, tudi optimalna. Seveda pa bi bilo na tako majhnem grafu preprosteje preveriti vse možne permutacije poti že na začetku (teh je $(4-1)! = 6$). V večjih grafih pa bo ta postopek imel veliko manjše število korakov kot naiven eksaktni algoritem (2-Opt ima namreč kubično časovno zahtevnost, naivni pa faktorialno).

Slika 5.17: Algoritem najbližjega soseda.

Input: points
Output: sorted_points

```
sorted_points = [points.first]
points.RemoveFirst()
while points ≠ ∅ do
    nearest = null
    for p ∈ points do
        if |p − sorted_points.last| < |nearest − sorted_points.last| then
            nearest = p
    sorted_points.Add(nearest)
    points.Remove(nearest)
```

Slika 5.18: Optimizacija 2-Opt.

Input: points

```
points.RemoveFirst()
steps = 0
while steps++ < 30 do
    if not 2-Opt(points) then
        break

boolean function 2-Opt(points)
    for i ∈ range [1..points.size-1] do
        for j ∈ range [i+1..points.size] do
            current = Invert(points, i, j)
            if TotalDistance(current) < TotalDistance(points) then
                points = current
                return true
    return false

// Invert subpath between indices i and j, inclusive
array function Invert(points, i, j)
    result=[]
    for k ∈ range [1..i-1] do
        result.Add(points[k])
    for k ∈ range [j downto i] do
        result.Add(points[k])
    for k ∈ range [j+1..points.size] do
        result.Add(points[k])
    return result
```

Poglavje 6

Rezultati

V tem poglavju bomo izmerili uspešnost, s katero posamezni agenti opravijo svojo nalogo in nato rezultate komentirali.

Za večjo zanesljivost in točnost meritev opravimo meritve pri vsakem od agentov večkrat in kot končno mero uspeha za primerjavo vzamemo povprečje meritev.

6.1 Prvi scenarij

V prvem scenariju izvedemo meritve za vsakega agenta 10-krat. Merimo število točk, torej število zadetih tarč. Čas izvajanja scenarija je 2 minuti.

6.1.1 Meritve

Rezultati meritev prvega scenarija so v tabeli 6.1. Std. odklon predstavlja standardni odklon, izračunan po formuli $\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$, kjer je N število meritev, $\overline{x}$ pa povprečje.

6.1.2 Analiza rezultatov

Agenti so se obnašali po pričakovanjih.

	SRA	**SBA**	**MBA**
Št. točk	0,6	14	57
Std. odklon	0,52	1,22	6,8

Tabela 6.1: Povprečna števila točk agentov v prvem scenariju.

Preprost odzivni agent (SRA) se je odrezal še nekoliko bolje, kot smo pričakovali, in je v šestih od desetih meritev uspel zadeti tarčo. Glede na preprostost, s katero je bil realiziran, je to kar dober dosežek. Ne prikazuje sicer še konkretne inteligence, vseeno pa kaže vsaj nekakšen začetek le-te. Akcije, ki jih izbira, povečajo pričakovano mero uspeha v primerjavi z agentom, ki bi jih izbiral povsem naključno. Precej visok standardni odklon je posledica naključnosti delovanja in pa binarnosti ocene pri tako slabem obnašanju (če je zadel tarčo, je dobil 1 točko, sicer pa 0 točk, zato meritve sorazmeroma močno odstopajo od povprečja).

Agent s stanji (SBA) se je v okolju obnašal že precej dobro. K temu je pripomoglo dejstvo, da je glede na trenutno zaznavo že maksimiziral pričakovano mero uspeha. Predznanja o scenariju pa še ni uporabljal, torej je nekakšen mejni kamen na polovici poti do popolne inteligence agenta. Nizek standardni odklon je dosegel, ker deluje vedno skoraj enako, ne glede na položaje tarč.

Agent s spominom (MBA) je uspešno upošteval tudi predznanje in si zapomnil vrstni red tarč, v katerem so se pojavljale. Kot vidimo, se je v scenariju obnesel daleč najbolje. Težko pa bi iz mere uspeha rekli, da je bil v scenariju približno štirikrat boljši od agenta s stanji. Agentu s stanji in preprostemu odzivnemu agentu namreč pričakovana mera uspeha raste linearno s časom scenarija, agentu s spominom pa ne (oz. mu raste z dvema različnima hitrostma), saj se najprej obnaša le tako dobro kot agent s stanji, ko pa enkrat do izraza pride spomin, pa se njegovo vedenje močno izboljša. Zadnji agent je v tem scenariju že primer bolj ali manj popolne inteligence, saj tako glede na predznanje kot tudi na zgodovino zaznav maksimizira pričakovano mero uspeha. Ima nekoliko višji standardni odklon kot prejšnji, ker je njegovo

obnašanje bolj odvisno od vrstnega reda tarč, ki je naključen.

Tudi če se analize lotimo bolj subjektivno, s prirejenim Turingovim testom – ali se je agent obnašal tako, da človek ne bi vedel, ali ga je vodil človek ali računalnik –, bi lahko rekli, da gre pri zadnjem že za zelo dober približek prave inteligence. Če izvzamemo nenaravno precizno premikanje kamere in nečloveške reflekse (oboje bi bilo enostavno popraviti, vendar z vidika inteligence to ni tako pomembno), je edina stvar, ki se nam zdi nenavadna, premikanje kamere v fazi iskanja tarče, ki je še vedno nekoliko preveč naključno.

6.2 Drugi scenarij

V drugem scenariju bomo izvedli dva seta meritev. Prvi bo v osnovnem okolju, drugi pa v okolju s premaknjenimi stojali in njihovimi okolicami.

V drugem izpustimo meritve preprostega odzivnega agenta, saj tako ali tako v scenariju deluje naključno in prestavitev okolja nanj nima konkretnega vpliva.

V prvem setu izvedemo po 5 meritev pri vsakem agentu, v drugem pa po 3, saj služi le za prikaz potrebe po avtonomnosti. Merimo čas izvajanja agenta in število vrnjenih sekir. Scenarij se sam konča po 7 minutah, na voljo pa je skupaj 15 sekir.

6.2.1 Meritve

Rezultati meritev osnovne oblike drugega scenarija so v tabeli 6.2. Tako za število točk kot za čas je v tabeli podan tudi standardni odklon. Mero uspeha izračunamo po formuli $10 \times povpr.\ št.\ točk - povpr.\ porabljen\ čas\ v\ minutah$.

6.2.2 Meritve prestavljenega scenarija

Rezultati meritev prestavljenega drugega scenarija so v tabeli 6.3.

	SRA	**GBA**	**SBA**	**MBA**	**AEA**	**UBA**
Št. točk	0,2	9,2	15	15	15	15
Std. odklon	0,45	1,92	0	0	0	0
Čas [m:ss]	7:00	4:31	5:43	4:21	4:30	3:13
Std. odklon [s]	0	13	49	61	34	15
Mera	-5	87,5	144,2	145,7	145,5	146,8

Tabela 6.2: Povprečna števila točk in časi agentov v drugem scenariju.

	GBA	**SBA**	**MBA**	**AEA**	**UBA**
Št. točk	1	8,7	10,7	15	15
Std. odklon	1	2,5	1,5	0	0
Čas [m:ss]	5:24	6:18	5:12	3:09	2:38
Std. odklon [s]	8,7	18	42	31	6,9
Mera	4,6	80,7	101,8	146,9	147,4

Tabela 6.3: Povprečna števila točk in časi agentov v premaknjenem drugem scenariju.

6.2.3 Analiza rezultatov

Preprost odzivni agent (SRA) je v okolju bolj ali manj odpovedal. Enkrat se mu je sicer posrečilo vrniti sekiro, v ostalih poskusih pa je 7 minut taval naokoli in ni opravil nič. Sicer uporablja zaznavo svojega položaja in časa, vendar pa si z njima ne pomaga pri povečevanju mere uspeha, pač pa akcije izbira naključno. Ker si torej ne pomaga niti s predznanjem niti z zaznavami, pri njem ne moremo govoriti o kakršnikoli inteligenci.

Preprost ciljno usmerjen agent (GBA) se je v prvem primeru obnesel precej dobro, vrnil je v povprečju približno dve tretjini sekir. Zaznave uporablja le na nižjem nivoju (za iskanje poti med točkami potrebuje informacijo o lastnem položaju), na višjem pa še ne. Delno pa uporablja predznanje, vendar brez možnosti prilagajanja. Zato lahko govorimo o približku inteligence le v prvem primeru scenarija, kjer je predznanje dejansko uporabil za povečanje

mere uspeha. V drugem primeru se je obnesel veliko slabše, tam o inteligenci ne moremo govoriti, saj pri izbiri akcij ni občutno boljši kot naključen agent.

Agent s stanji (SBA) se je v prvem primeru vedel že zelo dobro, saj mu je vedno uspelo vrniti vse sekire. Za povečanje pričakovanega števila sekir je uporabljal programerjevo predznanje, poleg tega pa tudi trenutno zaznavo. Tako je v prvem primeru že deloval z nekaj inteligence. V drugem pa je uporabljal le trenutno zaznavo brez kakršnekoli uporabe predznanja. Še vedno je torej tudi v drugem primeru šlo za nek približek inteligence, vendar precej slabši. Temu primeren je bil rezultat, vrnil je namreč le približno dve tretjini predmetov.

Agent s spominom (MBA) je deloval precej podobno agentu s stanji. Prav tako kot prejšnjemu mu je uspelo v prvem primeru vedno vrniti vse sekire, nekoliko pa je izboljšal povprečni čas. Do tega je prišlo, ker je namesto trenutne zaznave uporabljal celotno zaporedje zaznav in tako deloval bolj inteligentno v obeh primerih (čeprav v drugem še vedno veliko slabše). Tako je poleg povečevanja pričakovanega števila sekir zaznave uporabljal še za zmanjševanje porabljenega časa, torej je nekoliko optimiziral obe komponenti mere uspeha.

Agent z avtonomnim raziskovanjem (AEA) je bil prvi, ki je v obeh okoljih deloval enako dobro[1], kar kaže na visoko stopnjo avtonomnosti, ki je prejšnja dva nista imela. Vedno mu je uspelo vrniti vse sekire, je pa v prvem povprečno porabil več časa kot agent s spominom. Uporabljal je tako predznanje kot tudi zaporedje zaznav, da je maksimiziral število vrnjenih sekir, z zaznavami pa je tudi nekoliko zmanjšal pričakovani čas. Deloval je torej že precej inteligentno v obeh primerih scenarija.

Agent s funkcijo koristnosti (UBA) je deloval v obeh primerih daleč najbolje. Razlog za majhne relativne razlike v meri uspeha je seveda preprostost, s katero smo to definirali. Vedno je vrnil vse sekire, obenem pa je vedno

[1] V drugem je imel agent z avotonomnim raziskovanjem precej boljši čas, ker so premaknjene okolice na bolj odprtih mestih in posledično bolj pregledne, zato je hitreje zaključil s preiskovanjem. Zato je tudi prišlo do manjše razlike v primerjavi z agentom s funkcijo koristnosti.

potreboval občutno manj časa kot ostali. Tako predznanje kot zaporedje zaznav je uporabil za maksimiziranje števila vrnjenih sekir in minimiziranje porabljenega časa. V scenariju je torej bil že izredno dober približek popolne inteligence.

Če izvzamemo omejitev kamere pogleda na vodoravno lego in nenaravno vrtenje vedno za točno 360°, je agent tudi subjektivno gledano deloval zelo inteligentno. Nekoliko nas zmoti le pretirano pokrivanje prostora. Ljudje si namreč zelo dobro in brez težav zapomnimo (vsaj približno), katere dele sveta smo že videli (pregledali) in katerih ne, za agenta pa problem gradnje takega modela pregledanosti ni enostaven, zato smo uporabili pristop, kot smo ga (vnaprejšnja izbira točk, vrtenje na mestu), ki izpade nekoliko nečloveški.

Poglavje 7

Zaključek

V diplomskem delu smo med sabo primerjali delovanje različnih agentov umetne inteligence v testnih scenarijih, ki sta predstavljala situacije, s katerimi se srečamo v svetu igre. Prikazali smo tako uporabnost preprostejših agentov kot tudi potrebo po nadgradnji na bolj kompleksne.

Največjo razliko v obnašanje agentov je prinesla ciljna usmerjenost, ki agentom doda avtonomnost in zmožnost prilagajanja tako na spremembe v parametrih dane naloge kot tudi na dinamične spremembe, torej tiste, ki se zgodijo med samim delovanjem agenta. Tu smo videli razliko med predznanjem, ki je na voljo programerju ob implementaciji agenta in predznanjem, ki je na voljo agentu ob zagonu scenarija – slednje zahteva avtonomnost, če ga želimo dobro uporabiti.

Delitev agentov, ki smo jo opisali, je od tiste, ki jo predlagata Russel in Norvig, odstopala v tem, da smo definirali še agenta s stanji. Ta je pravzaprav podvrsta preprostega odzivnega agenta, vendar pa je dovolj pomemben, da ga omenjamo posebej. Kot smo videli, nam ta pristop omogoča zelo preprosto in kratko implementacijo delovanja, ki bi v osnovni verziji preprostega odzivnega agenta zahtevalo globoko gnezdeno in zapleteno kodo. Zato smo ta pristop uporabili v vseh implementiranih agentih (razen v preprostih odzivnih), uporablja pa se zelo pogosto tudi v umetni inteligenci v igrah. V implementiranih agentih v drugem scenariju pa smo nekoliko odstopili od te

delitve in agentu dodajali nadgradnje v drugačnem vrstnem redu. Russel in Norvig delitev opišeta kot nadgrajevanje zmožnosti, mi pa smo npr. ciljno usmerjenost implementirali pred spominom.

V obeh scenarijih smo z nadgrajevanjem agentov na koncu dobili že zelo dober približek inteligentnega obnašanja. Nekaj popravkov bi lahko še izvedli z vidika subjektivne človeškosti agentov, sicer pa so dane naloge opravili dobro.

Nekaj težav smo imeli z Unityjevim NavMeshom, ki, tudi ob dodatku t.i. HeightMesha, deluje nekoliko slabše na zelo razgibanem terenu (kot smo ga imeli v drugem scenariju), poleg tega pa ima agent na njem včasih težave z zelo kratkimi premiki, zato se je občasno zataknil. Ker je to težava z uporabljenim orodjem, ne pa v naši implementaciji, je elegantno in robustno odpravljanje le-te precej težek problem, ki ga lahko vzamemo kot predlog nadaljnjega dela.

Bolj povezano s temo te naloge pa bi lahko nadaljevali še z nepoznanim in večagentnim scenarijem, v katerih je pogosto zaželena (ali celo potrebna) zadnja nadgradnja, ki je v tej nalogi nismo implementirali – učljiv agent. Primeren scenarij za obe lastnosti bi bil na primer izsek igre RTS, kjer agent ne bi vnaprej vedel, katere enote so močne proti katerim (npr. suličarji so posebej močni proti konjenici). Tak scenarij bi bil tako nepoznan kot tudi večagenten, ne bi pa imel takega števila agentov, da bi bil učljiv agent prezahteven za računske vire. Tega scenarija bi se lahko lotil v okolju SEPIA [33], razvitem prav za namen raziskovanja agentov v igrah.

Nazadnje pa bi se lahko lotili še postopkov optimizacije delovanja agentov in dosega kompromisa med želeno inteligenco agentov in uporabo računskih virov, ki jih imamo na voljo. Ta dva tipa postopkov sta potrebna v scenarijih z velikim številom agentov, še posebej pa v igrah – tam namreč vse agente večinoma upravlja en računalnik, medtem ko imamo v splošnem v večagentnih scenarijih z velikim številom agentov pogosto za upravljanje leteh na voljo tudi večje število računalniških enot.

Literatura

[1] Counter-Strike: Global Offensive. [Online at www.counter-strike.net/ ; accessed 13-September-2015].

[2] Dragon Age: Origins. [Online at http://dragonage.bioware.com/dao/ ; accessed 13-September-2015].

[3] GitHub Repository. [Online at https://github.com/MatejVitek/Intelligent-Agents ; accessed 13-September-2015].

[4] GNU General Public License v3. [Online at http://www.gnu.org/licenses/gpl-3.0.html ; accessed 13-September-2015].

[5] Grand Theft Auto: San Andreas. [Online at http://www.rockstargames.com/sanandreas/ ; accessed 13-September-2015].

[6] Pong. [Online at http://www.ponggame.org/ ; accessed 13-September-2015].

[7] The Elder Scrolls V: Skyrim. [Online at http://www.elderscrolls.com/skyrim/ ; accessed 13-September-2015].

[8] The Last of Us. [Online at http://www.thelastofus.playstation.com/index.php#1 ; accessed 13-September-2015].

[9] Unity3D. [Online at https://unity3d.com/ ; accessed 13-September-2015].

[10] Unity3D Asset Store. [Online at `https://www.assetstore.unity3d.com/en/` ; accessed 13-September-2015].

[11] Unity3D Navigation Manual. [Online at `http://docs.unity3d.com/Manual/Navigation.html` ; accessed 13-September-2015].

[12] Unity3D System Requirements. [Online at `https://unity3d.com/unity/system-requirements` ; accessed 13-September-2015].

[13] Y. Al-Saawy, A. Al-Ajlan, K. Aldrawiesh, and A. Bajahzer. The Development of Multi-agent System Using Finite State Machine. In *New Trends in Information and Service Science (NISS), International Conference on*, pages 203–206. IEEE, 2009.

[14] P. Bailis, A. Fachantidis, and I. Vlahavas. Learning to play Monopoly: A Reinforcement Learning approach. In *Proceedings of the 50th Anniversary Convention of The Society for the Study of Artificial Intelligence and Simulation of Behaviour*. AISB, 2014.

[15] E. Bonomi and J. L. Lutton. The N-city travelling salesman problem: Statistical mechanics and the Metropolis algorithm. *SIAM review*, 26(4):551–568, 1984.

[16] C. Brandao, L. P. Reis, and A. P. Rocha. Evaluation of Embodied Conversational Agents. In *Information Systems and Technologies (CISTI), 8th Iberian Conference on*, pages 1–6, June 2013.

[17] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[18] R. Conte. *Rational, Goal-Oriented Agents*. Springer New York, 2012.

[19] S. K. Das. Modeling Intelligent Decision-Making Command and Control Agents: An Application to Air Defense. *Intelligent Systems, IEEE*, 29(5):22–29, Sept 2014.

[20] S. Dhakal and R. Chiong. A hybrid nearest neighbour and progressive improvement approach for Travelling Salesman Problem. In *Information Technology (ITSim), International Symposium on*, volume 1, pages 1–4, Aug 2008.

[21] J. Gemrot, C. Brom, and T. Plch. *Agents for Games and Simulations II.* Springer Berlin Heidelberg, 2011.

[22] J. Hagelback. Hybrid Pathfinding in StarCraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, accepted for publication, 2015.

[23] N. C. Hou, N. S. Hong, C. K. On, and J. Teo. Infinite Mario Bross AI using Genetic Algorithm. In *Sustainable Utilization and Development in Engineering and Technology (STUDENT), Conference on*, pages 85–89. IEEE, 2011.

[24] D. M. Kuiper and R. Z. Wenkstern. Agent vision in multi-agent based simulation systems. *Autonomous Agents and Multi-Agent Systems*, 29(2), 2015.

[25] V. Lisy, M. Jakob, J. Tozicka, and M. Pechoucek. Utility-based model for classifying adversarial behaviour in multi-agent systems. In *Computer Science and Information Technology (IMCSIT), International Multiconference on*, pages 47–53. IEEE, 2008.

[26] K. S. Løland. Intelligent agents in computer games. Master's thesis, Norwegian University of Science and Technology, Norway, 2008.

[27] I. Mavroidis, I. Papaefstathiou, and D. Pnevmatikatos. Hardware Implementation of 2-Opt Local Search Algorithm for the Traveling Salesman Problem. In *Rapid System Prototyping (RSP), 18th IEEE/IFIP International Workshop on*, pages 41–47, May 2007.

[28] E. Norling. *Agents for Games and Simulations.* Springer Berlin Heidelberg, 2009.

[29] P. G. Patel, N. Carver, and S. Rahimi. Tuning computer gaming agents using Q-learning. In *Computer Science and Information Systems (FedCSIS), Federated Conference on*, pages 581–588. IEEE, 2011.

[30] P. Peer. Skripta za predmet Tehnologija iger in navidezna resničnost, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani. [Online at `https://ucilnica.fri.uni-lj.si/pluginfile.php/10077/mod_resource/content/0/Predavanja/TINR_skripta_predavanj_5.pdf` ; accessed 13-September-2015], 2010.

[31] J. Rosmann, N. Hempe, and P. Tietjen. A flexible model for real-time crowd simulation. In *Systems, Man and Cybernetics (SMC), International Conference on*, pages 2085–2090. IEEE, 2009.

[32] S. Russel and P. Norvig. *Artificial Intelligence: A modern approach.*

[33] S. Sosnowski, T. Ernsberger, F. Cao, and S. Ray. SEPIA: A Scalable Game Environment for Artificial Intelligence Teaching and Research. In *Educational Advances in Artificial Intelligence, Fourth Association for the Advancement of Artificial Intelligence Symposium on*. AAAI Publications, 2013.

[34] W. van Toll, A. F. Cook, and R. Geraerts. Navigation meshes for realistic multi-layered environments. In *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, pages 3526–3532, Sept 2011.

[35] D. Wang and A. H. Tan. Creating Autonomous Adaptive Agents in a Real-Time First-Person Shooter Computer Game. *Computational Intelligence and AI in Games, IEEE Transactions on*, 7(2):123–138, June 2015.

[36] W. Zeng and R. L. Church. Finding shortest paths on real road networks: the case for A*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009.